

21天学编程系列

21天学通 Python (第2版)

—— 刘凌霄 郝宁波 吴海涛 编著 ——

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

本书全面、系统、深入地讲解了 Python 编程基础语法与高级应用。在讲解过程中，通过大量实际操作的实例将 Python 语言知识全面、系统、深入地呈现给读者。此外，本书配有大量微课，使用手机扫描书中的二维码即可在线观看，便于读者通过分析实例、运行实例，尽快熟悉 Python 编程，在学习遇到问题时，也可以作为参考。

本书内容共分 3 篇。第 1 篇介绍 Python 语言的特点、安装、语法基础、程序流程控制、面向过程的编程方法、面向对象的编程方法、程序异常的处理；第 2 篇介绍 Python 语言中的包与模块、迭代器、生成器、装饰器、上下文管理等进阶语法，同时介绍了使用 Python 标准库中的文件系统处理、图形化界面编程、正则表达式、网络编程、多进（线）程编程、数据库编程，还介绍了运用第三方库的 Web 网站编程、图片处理；第 3 篇通过两个案例介绍 Python 的综合编程技术。

本书内容涉及面广，从基本操作到高级技术及综合案例，涉及 Python 语言的基础语法和编程特性，而且实例实用、丰富，尤其适合广大编程初学者自学，也适合对 Python 语言感兴趣的爱好者作为参考用书。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

21 天学通 Python / 刘凌霄，郝宁波，吴海涛编著. —2 版. —北京：电子工业出版社，2018.2

（21 天学编程系列）

ISBN 978-7-121-33349-1

I. ①2… II. ①刘… ②郝… ③吴… III. ①软件工具—程序设计 IV. ①TP311.561

中国版本图书馆 CIP 数据核字(2017)第 320417 号

责任编辑：牛 勇

印 刷：三河市良远印务有限公司

装 订：三河市良远印务有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×1092 1/16 印张：21.5 字数：578 千字

版 次：2016 年 1 月第 1 版

2018 年 2 月第 2 版

印 次：2018 年 2 月第 1 次印刷

定 价：59.80 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888，88258888。

质量投诉请发邮件至 zltz@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

前言

千里之行，始于足下！

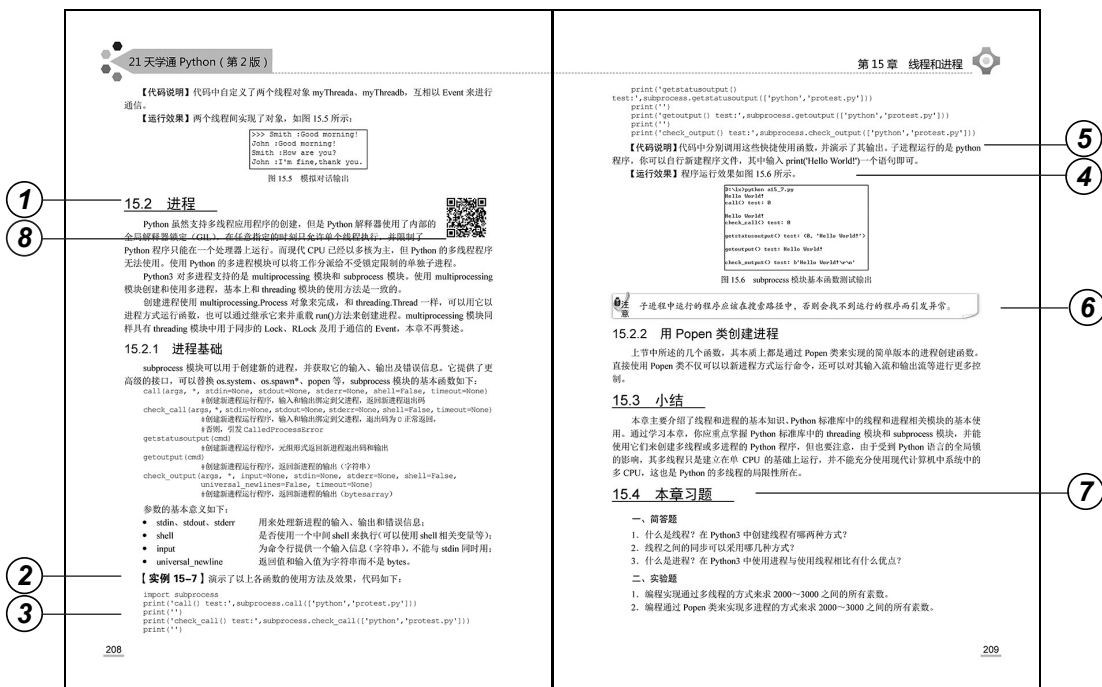
——老子

“21 天学编程系列”自 2009 年 1 月上市以来一直受到广大读者的青睐。该系列中的大部分图书从一上市就登上了编程类图书销售排行榜的前列，很多大、中专院校也将该系列中的一些图书作为教材使用，目前这些图书已经多次印刷、改版。可以说，“21 天学编程系列”是自 2009 年以来，国内原创计算机编程图书最有影响力的品牌之一。

本书有何特色

1. 细致体贴的讲解

为了让读者更快地上手，本书特别设计了适合初学者的学习方式，用准确的语言总结概念、用直观的图示演示过程、用详细的注释解释代码、用形象的比方帮助记忆。效果如下图所示。



① 知识点介绍 准确、清晰是其显著特点，一般放在每一节开始的位置，让零基础的读者了解相关概念，顺利入门。

② 实例 书中出现的完整实例，以章节顺序编号，便于检索和循序渐进地学习、实践，放在每节知识点介绍之后。

③ 示例代码 与实例编号对应,层次清楚、语句简洁、注释丰富,体现了代码优美的原则,有利于读者养成良好的代码编写习惯。对于大段程序,均在每行代码前设定编号,便于学习。

④ 运行效果 对实例给出运行结果和对应图示,帮助读者更直观地理解示例代码。

⑤ 代码说明 将实例代码中的关键代码行逐一解释,有助于读者掌握相关概念和知识。

⑥ 贴心的提示 为了便于读者阅读,全书还穿插着一些技巧、提示等小贴士,体例约定如下。

- 提示:通常是一些贴心的提醒,让读者加深印象,提供建议或者解决问题的方法。
- 注意:提出学习过程中需要特别注意的一些知识点和内容,或者相关信息。
- 警告:对操作不当或理解偏差将会造成的灾难性后果给出警示,以加深读者印象。

⑦ 习题 每章最后提供专门的测试习题,供读者检验所学知识是否牢固掌握。

⑧ 微课 使用手机 App (例如微信) 扫描二维码,可在线观看配套教学微课。

在本书中,所有的内容是基于 Python 3.x 实现的。

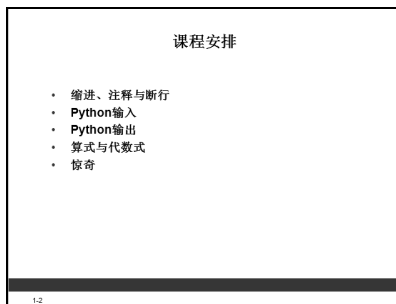
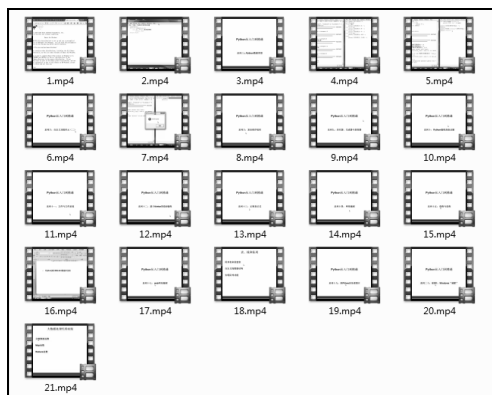
2. 实用超值的配套资源

为了帮助读者比较直观地学习,本书提供超值配套资源,内容包括多媒体视频、电子教案(PPT)和实例源代码等。

使用浏览器访问本书页面(<http://www.broadview.com.cn/33349>),可在“下载资源”处查看和下载本书配套资源文件。

• 多媒体视频

本书配有长达 10 小时的教学视频,讲解关键知识点界面操作和书中的一些综合练习题。作者亲自配音、演示,手把手教会读者使用。

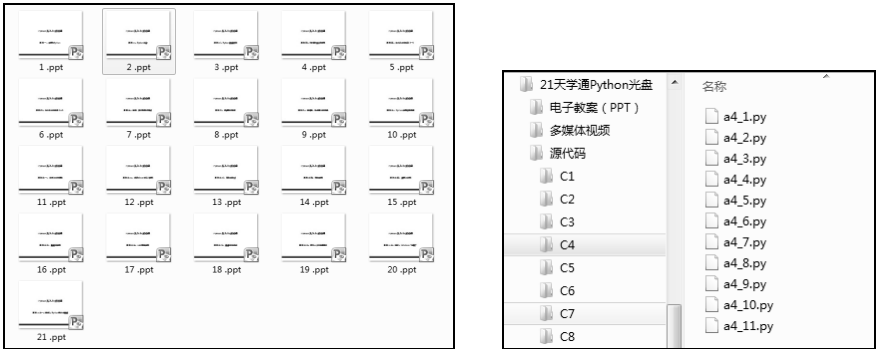


• 电子教案(PPT)

本书可以作为高校相关课程的教材或课外辅导书,所以作者特别为本书制作了电子教案(PPT),以方便老师教学使用。

• 源代码

本书附赠实例“源代码”。



3. 提供完善的技术支持

本书的技术支持论坛为 <http://www.rzchina.net>，读者可以在上面提问交流。另外，论坛上还有一些小的教程、视频动画和各种技术文章，可帮助读者提高开发水平。

推荐的学习计划

本书作者在长期从事相关培训或教学实践过程中，归纳了最适合初学者的学习模式，并参考了多位专家的意见，为读者总结了合理的学习时间分配方式，列表如下：

推荐时间安排		自学目标（框内打钩表示已掌握）		难度指数
第1周	第1天	Python 的特点 在 Windows 下和 Linux 下安装 Python 的流程 编译和运行 Python 程序	<div><input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/></div>	★★
	第2天	Python 的基础语法 Python 最简单的键盘输入与屏幕输出 用 Python 进行算式的计算	<div><input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/></div>	★★
	第3天	Python 的简单数据类型 Python 的结构数据类型 内置常量与逻辑运算符、比较运算符 序列的使用	<div><input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/></div>	★★★
	第4天	if 选择执行语句 for 和 while 循环执行语句 推导或内涵	<div><input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/></div>	★★★★
	第5天	如何声明函数 如何调用函数 函数的各种应用 匿名函数的使用 Python 常用的内建函数	<div><input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/></div>	★★★★★
	第6天	了解面向对象编程 学会定义和使用类 类的属性和方法 类的继承	<div><input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/></div>	★★★

续表

推荐时间安排		自学目标（框内打钩表示已掌握）	难度指数	
第 1 周	第 7 天	语法错误和异常的概念	<input type="checkbox"/>	★★★
		用 try 语句捕获异常	<input type="checkbox"/>	
		常见异常的处理	<input type="checkbox"/>	
		自定义异常	<input type="checkbox"/>	
		使用 pdb 调试 Python 程序	<input type="checkbox"/>	
第 2 周	第 8 天	模块的概念、用法，以及对编程的影响	<input type="checkbox"/>	★★★★
		包的概念及使用	<input type="checkbox"/>	
	第 9 天	自定义迭代器及内置迭代器	<input type="checkbox"/>	★★★★
		生成器的创建与协程	<input type="checkbox"/>	
		装饰器概念、应用函数装饰器与应用类装饰器	<input type="checkbox"/>	
	第 10 天	函数与命名空间	<input type="checkbox"/>	★★★
		闭包、闭包实现延迟求值、闭包实现泛型函数	<input type="checkbox"/>	
		上下文管理器	<input type="checkbox"/>	
		用字符串操作对象属性	<input type="checkbox"/>	
		用字典构造分支程序	<input type="checkbox"/>	
		重载与多态	<input type="checkbox"/>	
	第 11 天	文件函数与文件的读写操作	<input type="checkbox"/>	★★★
		处理文件中的数据	<input type="checkbox"/>	
	将 py 程序文件打包为 exe	<input type="checkbox"/>		
第 12 天	GUI 的概述与 tkinter 图形化库	<input type="checkbox"/>	★★★★★	
	tkinter 组件的使用：按钮、文本框、标签、菜单、单选框等	<input type="checkbox"/>		
	响应操作事件	<input type="checkbox"/>		
标准对话框与自定义对话框	<input type="checkbox"/>			
第 13 天	正则表达式基本元字符与常用的正则表达式	<input type="checkbox"/>	★★★	
	Python 的 re 正则模块	<input type="checkbox"/>		
	分组匹配与匹配对象使用	<input type="checkbox"/>		
	正则表达式的具体应用案例	<input type="checkbox"/>		
第 14 天	网络编程预备知识	<input type="checkbox"/>	★★★★	
	用 socket 建立客户端与服务器	<input type="checkbox"/>		
	用 socketserver 建立基本的服务器	<input type="checkbox"/>		
	使用 http、urllib 标准库	<input type="checkbox"/>		
	用 poplib 与 smtplib 处理邮件	<input type="checkbox"/>		
	用 ftplib 访问 FTP 服务器	<input type="checkbox"/>		
第 3 周	第 15 天	线程、进程基础	<input type="checkbox"/>	★★★★★
		用 threading 模块进行多线程编程	<input type="checkbox"/>	
		用 subprocess 模块多进程编程	<input type="checkbox"/>	
			<input type="checkbox"/>	

续表

推荐时间安排		自学目标（框内打钩表示已掌握）	难度指数	
第 3 周	第 16 天	Python 数据库 API 基础	<input type="checkbox"/>	★★★★★
		Python 操作 SQLite3	<input type="checkbox"/>	
		Python 操作 MariaDB	<input type="checkbox"/>	
		Python 操作 MongoDB	<input type="checkbox"/>	
		ORM 的框架 mongoengine	<input type="checkbox"/>	
	第 17 天	Flask 框架安装及应用	<input type="checkbox"/>	★★★★★
		Tornado 框架安装	<input type="checkbox"/>	
	第 18 天	用 Python 操作表	<input type="checkbox"/>	★★★★★
		用 Python 操作栈	<input type="checkbox"/>	
		用 Python 操作队列	<input type="checkbox"/>	
		用 Python 操作树	<input type="checkbox"/>	
		用 Python 操作图	<input type="checkbox"/>	
		用 Python 进行查找	<input type="checkbox"/>	
用 Python 进行排序		<input type="checkbox"/>		
第 19 天	第三方 Pillow 库	<input type="checkbox"/>	★★★★★	
第 20 天	综合案例	<input type="checkbox"/>	★★★★	
第 21 天	综合案例	<input type="checkbox"/>	★★★★★	

本书适合哪些读者阅读

本书非常适合以下人员阅读：

- 希望学习和使用 Python 语言的新手；
- 迫切希望全面且深入地学习 Python 语言的程序人员；
- 具备一定编程经验的程序员；
- 希望了解和使用 Python 语言，并以它作为第二语言的编程人员。

本书作者

本书主要由刘凌霄、郝宁波、吴海涛编写，参与编写工作的还有：张昆、张友、赵桂芹、张金霞、张增强、刘桂珍、陈冠军、魏春、张燕、孟春燕、项宇峰、李杨坡。由于水平有限，书中难免存在疏漏和不足之处，恳请广大读者和专家批评、指正。

目 录

第 1 篇 Python 编程基础

第 1 章 编程与 Python	1
1.1 什么是编程	1
1.1.1 硬件与软件	1
1.1.2 编程语言	2
1.1.3 编程与调试	3
1.2 选择 Python 的理由	4
1.2.1 Python 是免费的自由软件	4
1.2.2 Python 是跨平台的	4
1.2.3 Python 功能强大	4
1.2.4 Python 清晰优雅	5
1.3 安装 Python	5
1.3.1 在 Windows 下安装 Python	6
1.3.2 在 Linux 下安装 Python	7
1.4 选择开发工具	9
1.4.1 Python 自带开发工具: IDLE	9
1.4.2 文本编辑器: Emacs	10
1.4.3 Python 开发工具: PythonWin	12
1.5 编辑和运行 Python 程序	15
1.5.1 你好, Python	15
1.5.2 运行程序文件	15
1.5.3 交互式运行 Python	16
1.6 小结	17
1.7 本章习题	17
第 2 章 Python 起步	18
2.1 Python 语法基础	18
2.1.1 缩进分层	18
2.1.2 代码注释	19
2.1.3 断行	19
2.2 Python 输入/输出	20
2.2.1 接收键盘/输入	20
2.2.2 显示处理结果	21
2.3 用 Python 计算	21
2.3.1 算式与代数式运算	22



2.3.2 惊奇	23
2.4 小结	24
2.5 本章习题	24
第 3 章 Python 数据类型	26
3.1 Python 简单数据类型	26
3.1.1 字符串 (str)	26
3.1.2 整数 (int)	29
3.1.3 浮点数 (float)	30
3.1.4 类型转换	30
3.2 字符串进阶	31
3.2.1 原始字符串	31
3.2.2 格式化字符串	32
3.2.3 中文字符串处理	32
3.3 标志符与赋值号	33
3.3.1 标志符	33
3.3.2 赋值号 “=”	33
3.4 Python 结构数据类型	34
3.4.1 列表 (list)	34
3.4.2 元组 (tuple)	36
3.4.3 字典 (dict)	36
3.5 内置常量与逻辑运算符、比较运算符	38
3.5.1 常用内置常量	38
3.5.2 Python 中逻辑运算符	38
3.5.3 Python 中比较运算符	39
3.5.4 Python 中其他逻辑操作符	39
3.6 序列	40
3.6.1 序列切片	40
3.6.2 序列内置操作	41
3.7 小结	42
3.8 本章习题	42
第 4 章 控制语句执行流程	44
4.1 用 if 选择执行语句	44
4.1.1 if 基础	44
4.1.2 if 语句的嵌套	47
4.2 用 for 循环执行语句	49
4.2.1 for 基础	49
4.2.2 for 语句与 break 语句、continue 语句	49
4.2.3 for 语句与 range() 函数	51
4.2.4 for 语句与内置迭代函数	52
4.3 用 while 循环执行语句	53
4.3.1 while 基础	53
4.3.2 增量赋值运算符	54

4.4 推导或内涵 (list comprehension)	55
4.4.1 推导基础	55
4.4.2 推导进阶	55
4.5 小结	56
4.6 本章习题	56
第5章 自定义功能单元 (一)	58
5.1 使用函数	58
5.1.1 声明函数	58
5.1.2 调用函数	59
5.2 深入函数	60
5.2.1 默认值参数	60
5.2.2 参数传递	62
5.2.3 可变数量参数传递	62
5.2.4 拆解序列的函数调用	65
5.2.5 函数调用时参数的传递方法	65
5.3 变量的作用域	67
5.4 使用匿名函数 (lambda)	68
5.5 Python 常用内建函数	69
5.6 小结	70
5.7 本章习题	70
第6章 自定义功能单元 (二)	72
6.1 面向对象编程概述	72
6.1.1 万物皆对象	72
6.1.2 对象优越性	73
6.1.3 类和对象	73
6.2 定义和使用类	73
6.2.1 定义类	73
6.2.2 使用类	74
6.3 类的属性和方法	75
6.3.1 类的方法	75
6.3.2 类的属性	77
6.3.3 类成员方法与静态方法	79
6.4 类的继承	80
6.4.1 类的继承	80
6.4.2 多重继承	81
6.4.3 方法重载	83
6.5 小结	83
6.6 本章习题	83
第7章 错误、异常和程序调试	85
7.1 语法错误	85
7.2 异常的处理	86



7.2.1	异常处理的基本语法	86
7.2.2	Python 主要的内置异常及其处理	88
7.3	手工抛出异常	90
7.3.1	用 raise 手工抛出异常	90
7.3.2	assert 语句	91
7.3.3	自定义异常类	92
7.4	用 pdb 调试程序	93
7.4.1	调试语句块函数	93
7.4.2	调试函数	94
7.5	测试程序	95
7.5.1	用 testmod 函数测试	95
7.5.2	用 testfile 函数测试	96
7.6	小结	97
7.7	本章习题	97
 第 2 篇 Python 编程高阶 		
第 8 章	复杂程序组织	99
8.1	模块	99
8.1.1	模块概述	99
8.1.2	自己编写模块	100
8.1.3	模块位置	101
8.1.4	__pycache__ 目录	102
8.1.5	具有独立运行能力的模块	102
8.2	包	103
8.2.1	包概述	103
8.2.2	包详解	104
8.3	Python 常用标准库简介	105
8.3.1	数学类模块	105
8.3.2	日期与时间类	106
8.4	小结	106
8.5	本章习题	106
第 9 章	迭代器、生成器与装饰器	108
9.1	迭代器	108
9.1.1	迭代器概述	108
9.1.2	自定义迭代器	109
9.1.3	内置迭代器工具	109
9.2	生成器	112
9.2.1	生成器创建	112
9.2.2	深入生成器	113
9.2.3	生成器与协程	114
9.3	装饰器	115
9.3.1	装饰器概述	115

9.3.2 装饰函数	115
9.3.3 装饰类	116
9.4 小结	117
9.5 本章习题	118
第 10 章 Python 进阶话题	119
10.1 函数与命名空间	119
10.2 闭包及其应用	120
10.2.1 闭包概述	120
10.2.2 闭包与延迟求值	121
10.2.3 闭包与泛型函数	121
10.3 上下文管理器	122
10.4 用字符串操作对象属性	124
10.5 用字典构造分支程序	125
10.6 重载类的特殊方法	126
10.7 鸭子类型 (duck typing) 与多态	127
10.8 小结	128
10.9 本章习题	128
第 11 章 文件与文件系统	130
11.1 文件操作基础	130
11.1.1 open()函数	130
11.1.2 用 fileinput 操作文件	132
11.2 常用文件和目录操作	133
11.2.1 获得当前路径	133
11.2.2 获得目录中的内容	133
11.2.3 创建目录	134
11.2.4 删除目录	134
11.2.5 判断是否是目录	134
11.2.6 判断是否为文件	134
11.2.7 遍历某目录下的所有文件和目录	135
11.2.8 由文件名批量获取姓名和考号	135
11.2.9 批量文件重命名	136
11.3 编译为可执行文件	137
11.3.1 用 py2exe 生成可执行程序	137
11.3.2 用 cx_freeze 生成可执行文件	138
11.4 小结	140
11.5 本章习题	140
第 12 章 基于 tkinter 的 GUI 编程	141
12.1 GUI 概述	141
12.1.1 GUI 是什么	141
12.1.2 Python 编写 GUI 程序库	141
12.2 tkinter 图形化库简介	142



12.2.1	创建 GUI 程序第一步	142
12.2.2	创建 GUI 程序第二步	143
12.3	tkinter 组件	144
12.3.1	组件分类	144
12.3.2	布局组件	144
12.4	常用 tkinter 组件	145
12.4.1	按钮	145
12.4.2	文本框	147
12.4.3	标签	148
12.4.4	菜单	149
12.4.5	单选框和复选框	151
12.4.6	绘制图形	153
12.5	响应操作事件	155
12.5.1	事件基础	155
12.5.2	响应事件	157
12.6	对话框	159
12.6.1	标准对话框	159
12.6.2	自定义对话框	165
12.7	小结	166
12.8	本章习题	166
第 13 章	正则表达式	168
13.1	正则表达式基础	168
13.1.1	正则表达式概述	168
13.1.2	正则表达式基本元字符	168
13.1.3	常用正则表达式	170
13.2	re 模块	171
13.2.1	正则匹配搜索函数	171
13.2.2	sub()与 subn()函数	172
13.2.3	split()函数	173
13.2.4	正则表达式对象	173
13.3	分组匹配与匹配对象使用	177
13.3.1	分组基础	177
13.3.2	分组扩展	177
13.3.3	匹配对象与组的使用	178
13.3.4	匹配对象与索引使用	179
13.4	正则表达式应用示例	180
13.5	小结	182
13.6	本章习题	182
第 14 章	网络编程	183
14.1	网络编程基础	183
14.1.1	什么是网络	183
14.1.2	网络协议	183

14.1.3 地址与端口	184
14.2 套接字的使用	185
14.2.1 用 socket 建立服务器端程序	185
14.2.2 用 socket 建立客户端程序	186
14.2.3 用 socket 建立基于 UDP 协议的服务器与客户端程序	188
14.2.4 用 socketserver 模块建立服务器	189
14.3 urllib 与 http 包使用	190
14.3.1 urllib 和 http 包简介	190
14.3.2 用 urllib 和 http 包访问网站	193
14.4 用 poplib 与 smtplib 库收发邮件	194
14.4.1 用 poplib 检查邮件	194
14.4.2 用 smtplib 发送邮件	196
14.5 用 ftplib 访问 FTP 服务	198
14.5.1 ftplib 模块简介	198
14.5.2 使用 Python 访问 FTP	200
14.6 小结	202
14.7 本章习题	202
第 15 章 线程和进程	203
15.1 线程	203
15.1.1 用 threading.Thread 直接在线程中运行函数	203
15.1.2 通过继承 threading.Thread 类来创建线程	204
15.1.3 线程类 Thread 使用	204
15.2 进程	208
15.2.1 进程基础	208
15.2.2 用 Popen 类创建进程	209
15.3 小结	211
15.4 本章习题	211
第 16 章 数据库编程	212
16.1 Python 数据库应用程序接口	212
16.1.1 数据库应用程序接口概述	212
16.1.2 数据库游标的使用	213
16.2 Python 操作 SQLite3 数据库	213
16.2.1 SQLite3 数据库简介	214
16.2.2 SQLite3 数据库操作实例	214
16.3 Python 操作 MariaDB 数据库	216
16.3.1 MariaDB 数据库简介	217
16.3.2 建立 MariaDB 数据库操作环境	217
16.3.3 MariaDB 数据库操作实例	219
16.4 Python 操作 MongoDB 数据库	221
16.4.1 MongoDB 数据库简介	221
16.4.2 建立 MongoDB 数据库操作环境	221
16.4.3 MongoDB 数据库基础	222



16.4.4	MongoDB 数据库操作实例	225
16.4.5	用对象关系映射（ORM）工具操作 MongoDB 数据库	227
16.5	小结	230
16.6	本章习题	230
第 17 章	Web 网站编程	231
17.1	Web 网站编程概述	231
17.2	Flask Web 框架及其应用	232
17.2.1	Flask Web 框架简介	232
17.2.2	Flask Web 框架初识	232
17.2.3	URL 装饰器与 URL 参数传递	234
17.2.4	GET 与 POST 请求的参数传递	236
17.2.5	使用 cookie 与 session 跟踪客户	238
17.2.6	使用静态文件资源与页面文件	241
17.2.7	接收上传文件	242
17.2.8	在 Flask 框架中使用数据库	243
17.3	Tornado Web 框架及其应用	246
17.3.1	Tornado 框架简介	246
17.3.2	Tornado 框架初识	246
17.3.3	请求参数的获取	248
17.3.4	用 cookie 与安全 cookie 跟踪客户	250
17.3.5	URL 转向与静态文件资源	251
17.3.6	Tornado Web 框架应用举例	253
17.4	小结	258
17.5	本章习题	258
第 18 章	数据结构基础	260
18.1	表、栈和队列	260
18.1.1	用列表来创建表	260
18.1.2	自定义栈数据结构	261
18.1.3	实现队列功能	263
18.2	树和图	264
18.2.1	用列表构建树	264
18.2.2	实现二叉树类与遍历二叉树	265
18.2.3	用字典构建与搜索图	268
18.3	查找与排序	270
18.3.1	实现二分查找	270
18.3.2	用二叉树排序	272
18.4	小结	274
18.5	本章习题	274
第 19 章	用 Pillow 库处理图片	275
19.1	第三方 Pillow 库概述	275
19.1.1	安装第三方 Pillow 库	275

19.1.2	Pillow 库简介	275
19.1.3	Pillow 库处理图像基础	276
19.1.4	Image 模块中函数的使用	277
19.1.5	Image 模块中 Image 类的使用	280
19.1.6	使用 ImageChops 模块进行图片合成	285
19.1.7	使用 ImageEnhance 模块增强图像效果	288
19.1.8	使用 ImageFilter 模块的滤镜	289
19.1.9	使用 ImageDraw 模块画图	290
19.2	使用 Pillow 库处理图片举例	291
19.2.1	图片格式转换	291
19.2.2	批量生成缩略图	293
19.2.3	为图片添加 Logo	296
19.3	小结	300
19.4	本章习题	301
 第 3 篇 Python 编程实战 		
第 20 章	案例 1 做一个 Windows 上的 360 工具	302
20.1	案例背景	302
20.2	从创建图形化界面开始	303
20.2.1	创建基本图形化工作界面	303
20.2.2	响应菜单事件	305
20.3	清理垃圾文件	307
20.3.1	迭代目录	307
20.3.2	扫描垃圾文件	308
20.3.3	多线程加速	310
20.3.4	扫描所有磁盘	311
20.3.5	删除垃圾文件	313
20.4	搜索文件	315
20.4.1	搜索超大文件	315
20.4.2	按名称搜索文件	316
20.5	小结	317
第 21 章	案例 2 Python 搞定大数据	319
21.1	案例背景	319
21.1.1	大数据处理方式概述	319
21.1.2	处理日志文件	320
21.1.3	要实现的案例目标	321
21.2	分割日志文件	321
21.3	用 Map 函数处理小文件	323
21.4	用 Reduce 函数归集数据	325
21.5	小结	326

第 1 篇 Python 编程基础

第 1 章 编程与 Python

自从计算机诞生之日起，编程就是计算机相关工作的一部分，如今编程不再只是与计算机有关的工作，它已经渗透到社会生成的各个领域中去了，不管你是否从事 IT 行业，懂得点编程也是工作和生活的需要之一。那么，什么是编程呢？

如今的编程语言可以说是五花八门、百花齐放，发展至今已经有几十种计算机语言。从最初的机器语言，到今天的高级语言，让你选择的话，可能会无所适从。

Python 语言是一种开放源代码的、免费的跨平台语言，它既具有当今高级语言所具有的面向对象等特性，又具有清晰的结构、简洁的语法。Python 语言自身带有丰富而实用的标准库，还拥有大量的第三方开源库，能够应付各种场合。Python 语言可以使用 C/C++ 进行扩展，也可以嵌入到其他语言之中。所以，本书推荐的是 Python。

本章内容包括：

- 编程预备知识；
- Python 语言特点；
- 安装 Python 工作环境；
- Python 语言开发工具介绍；
- 建立和运行 Python 程序。

1.1 什么是编程

简单地说，编程就是安排计算机解决某个问题的方法、步骤。但要详细了解编程还需要你了解计算机的相关知识。



1.1.1 硬件与软件

计算机硬件是计算机中你能够触及到的部分，换句话说就是你能踢它一脚的东西。从信息处理的角度来看计算机，其结构如图 1.1 所示。

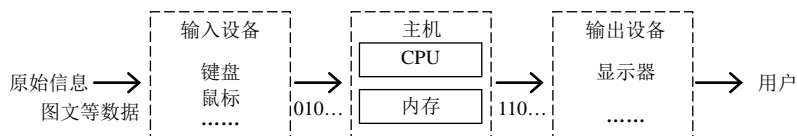


图 1.1 信息流与计算机硬件

如图 1.1 所示，信息由外部世界经由输入设备进入计算机，然后在主机（CPU 和内存）内进行处理，最后通过输出设备把信息处理的结果送出。在这个过程中，这些输入和输出设备根据信息的特点而表现不同，比如要输入字符，可以通过键盘进行录入。

最常见的输入设备就是键盘和鼠标，它是用户向计算机输入信息的主要设备，因此你所看到的计算机，几乎全都带有这两种设备。

输出设备对于计算机来说，也很重要。计算机只有具有了输出设备，人类才能了解它的运

行状态,获取信息处理的结果,而不同的输出设备以不同的形式输出信息处理结果。最常用的输出设备就是以图文形式输出信息的显示器和打印机,当然还包括音箱、绘图仪等。

计算机的本质是信息处理机,要处理信息就必须具备处理信息的硬件设备。类似于人的大脑,要处理信息就要有运算能力和存储能力。计算机主机中的 CPU (包括运算器和控制器) 就具有运算能力及控制能力;存储器就具有存储能力。

对于计算机来说,由其自身特点决定了它的信息处理和信息存储的形式,即使用二进制代码(只有 0 和 1)来表示所有的信息。外界所有的信息要进入计算机,都必须千方百计地转换为二进制代码,这些信息有文字、图片、声音、视频等各种形式。

计算机在进行信息处理时,必须能够按照人们的要求,通过一定的方式、方法或者步骤来进行。要实现这个目标,总不能每做一步都由人临时控制,因此,计算机就必须通过软件(程序)来进行控制。计算机输入信息、处理信息和输出信息都必须通过软件来进行控制。

由计算机硬件概念可推知,计算机软件是指你不能触及的部分,它实际上是对计算机进行控制的方式或步骤的描述。

计算机要想完成某个任务,就要知道完成这个任务所需的人为指定的步骤或方法。对于某个特定的问题,人们通过给定的指令序列来进行方法或步骤的描述,而这里的指令序列本质上应该是 CPU 指令的序列,也就是程序。

此外,要解决某一问题总是有一定方法的,这个方法的具体实现方式被称之为算法。就像解一道数学应用题,既可以使用列综合算式方法来完成,也可以通过列方程的方法来完成,而这些不同的算法最后达成的目标是相同的。

很显然,计算机能够运行的前提是必须同时具有硬件和软件。硬件是一切工作的基础,也是计算机赖以存在的物质基础;而软件是硬件工作必不可少的控制者。同时具有硬件系统和软件系统才能构成一个完整的计算机系统。

1.1.2 编程语言

编程语言在计算机诞生的那一天起就存在了,那个时候,人类自然会想到用计算机表示信息的方式(二进制代码 0 和 1)来描述对计算机的控制程序,这就是机器语言,它同时也是这台计算机 CPU 的内建命令集。由此也可以看出,机器语言其实就是人类用计算机本身的语言来完成对计算机的控制的,要想控制计算机,人类就要学会机器所使用的机器语言(用 0 和 1 描述)。

当然,可想而知:用一长串 0 和 1 来写程序控制计算机的话,效率肯定很低,既不容易理解,又很容易搞错。此外,不同类型的计算机其机器语言也不同,也就是说即使是为了解决同样的问题,在不同的计算机中,程序也不相同。

要想解决机器语言难学、易错的问题,第一步是不用 0 和 1 来写程序,因为那太难记、难写,又很容易被错写。那就用单词来代替命令代码(一串 0、1),因此诞生了汇编语言,汇编语言不是采用二进制代码来描述解决问题的步骤,而计算机只能识别二进制代码,所以运行时要先翻译为二进制代码的机器语言(专业术语叫编译),然后计算机才能识别和执行。

汇编语言虽然解决了易错、难记的问题,但还是离人类的语言太遥远了,如果能像对一个人说话一样来写程序那就最好了。可是计算机毕竟是机器,要实现这个目标还有很长的一段路要走。人们就使用了能够较为准确描述算法步骤的接近于人类语言和数学表示方法的形式来作为写程序的语言,即现代常用的编程语言——高级语言。常见的高级语言有 Python、C、Java、Perl、Erlang、LISP 等。

当然,高级语言也是计算机不能直接理解的,运行前必须进行编译,最终成为机器语言,



计算机才能理解和执行。高级语言的执行方式分为两种：一种是编译执行，即程序编写完成后直接将其编译为机器语言后执行；另一种是解释执行，即程序一边解释一边运行。比如，C 语言采取的是编译执行方式，而 Python 语言采取解释执行的方式。

要学习一门语言，首要任务就是掌握其语法及语义，其次就是熟练地使用它，最后是能够写出高效运行的程序，解决某个具体的问题。

1.1.3 编程与调试

编程就是根据要解决的某个具体问题，设计出相应的解决步骤（算法），根据所用语言的语法和语义，写出对应的命令序列。

由此可见，编程要完成的基本任务：

- 设计科学的算法；
- 写出正确的程序；
- 合理的人机交互。

1. 编程

(1) 设计科学的算法

解决某一具体的问题往往有多种不同的算法或步骤。所谓科学的算法，就是既要保证算法能在各种情况下正常地工作并且得到正确的处理结果，又要使程序能高效率运行。高效率运行的程序是要求它能够使用尽量少的系统资源在较短的时间内得出正确的处理结果。算法是程序的灵魂，只有设计出好的算法，才能写出高效率的程序。

(2) 写出正确的程序

是指运用指定语言的语法和语义写出实现设计算法的程序。一个具有语法错误的程序是不能正常运行的，当然也不会对信息进行处理，更谈不上得到处理结果；程序的逻辑错误会导致程序不会按照预先设计的算法进行信息处理，当然也不会得到正确的信息处理结果。

(3) 合理的人机交互

编程的最终目标是解决某个具体的问题，既需要信息的输入，又需要信息的输出。实现了友好的信息输入和输出，才能更好地解决问题。

然而，再高明的程序员写程序也不是一蹴而就的，写出的程序不可能没有一点语法错误或逻辑错误，所以编程要完成的第二方面的任务就是：

- 排除语法错误；
- 排除逻辑错误；
- 做好后期维护。

2. 调试

(1) 排除语法错误

对于编写的程序中的语法错误，只要尝试运行之后即可发现，一般来说，程序能正常运行起来，程序中就不会有语法错误了。

(2) 排除逻辑错误

若存在逻辑错误，程序依然可以正常运行，可能有时能得到正确的处理结果，有时则得不到。在排除这种逻辑错误时必须要对程序进行测试或调试，对于一个小型应用，可以采取罗列或穷举法来根据处理结果发现是否有逻辑错误，从而进一步排除错误。而对于大中型应用项目则需要专门测试，才能发现逻辑错误，从而将其排除。

(3) 做好后期维护

随着一个应用项目的实施,用户可能会提出一些改进意见或修改建议,程序员要进行及时修正与维护。

即便是最精通程序设计的程序员也可能写出不完美的代码,导致程序中存在着 bug 或错误,但大多数 bug 就是错误。如果程序不能够得到预期的运行结果,也不必惊慌失措,通过多次试运行和仔细测试一定可以使程序能够正常运行并得到预期结果。当遇到屡次修正不了的 bug 时,可以暂停一会儿,休息一下再回来工作,可能对调试者更有帮助。此外,程序指令的一小处改动,也可能会引入比你移除的 bug 更多的 bug。

1.2 选择 Python 的理由

Python 语言是一种高级语言,也是一种面向对象、解释型的程序设计语言,由 Guido van Rossum 于 1989 年底发明,其第一个公开发行人版发行于 1991 年,它遵循 GPL 协议,是源代码开放的软件。用户可以免费使用 Python 来编写程序,还可以阅读 Python 的源代码,甚至参与到 Python 的开发之中。

Python 有什么优点呢?下面来看一下。

1.2.1 Python 是免费的自由软件

Python 遵循 GPL 协议,是自由软件,这是 Python 流行的原因之一。用户使用 Python 进行开发和发布自己编写的程序不需要支付任何费用,不用担心版权问题,即使作为商业用途,Python 也是免费的。开源的自由软件正在成为软件行业的一种发展趋势,现在,很多商业软件公司也开始将自己的产品变为开源的,例如 Java。作为开源软件的 Python 将具有更强的生命力。

作为自由软件,最令人鼓舞的就是可以很方便地获取源代码。程序员通过阅读其源代码,可以发现其中的神奇之处,编写出更加高效的程序。

1.2.2 Python 是跨平台的

跨平台和良好的可移植性是 C 语言成为经典编程语言的关键,而 Python 正是用可移植的 ANSI C 编写的。这意味着 Python 也具有好的跨平台特性,也就是说,在 Windows 下编写的 Python 程序可以轻易地运行在 Linux 下。当然如果在 Python 程序中使用了 Windows 的某些特性(比如 COM),那就另当别论了。

Python 不仅能在 Windows、Linux 系统中运行,由于其开源本质,它已经被移植在许多平台上,包括 Linux、Windows、FreeBSD、Macintosh、Solaris、OS/2、Amiga、AROS、AS/400、BeOS、OS/390、z/OS、Palm OS、QNX、VMS、Psion、Acom RISC OS、VxWorks、PlayStation、Sharp Zaurus、Windows CE、PocketPC、Symbian 以及 Google 基于 Linux 开发的 Android 平台,可以看出,Python 不仅能运行在传统的 Server、PC 系统中,还能在正蓬勃发展的各类移动系统中运行。

1.2.3 Python 功能强大

也许 Python 强大的功能才是很多用户支持 Python 的最重要原因,从字符串处理到复杂的 3D 图形编程,Python 借助扩展模块都可以轻易完成。实际上 Python 的核心模块就已经提供了足够强大的功能,使用 Python 精心设计的内置模块可以完成许多功能强大的操作,更为重要的是 Python 还有丰富的、开源的第三方库,可以支持大量的不同应用。



1.2.4 Python 清晰优雅

Python 语言语法简单，写出的程序必须严格遵守其缩进规则，其语句块的标志就是由缩进来决定的，这使得 Python 程序格式清晰、易写、易读。Python 的设计哲学是“优雅”“明确”“简单”，而其开发时的指导思想是：对于一个特定的问题，只要有一种最好的方法来解决。在 Python 交互式环境中使用 `import this` 就可以显示出 Python 之禅（The Zen of Python, by Tim Peters），其内容如图 1.2 所示。

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

图 1.2 Python 之禅

前面最经典的几句译文分别是：

- 优美胜于丑陋。
- 明了胜于晦涩。
- 简洁胜于复杂。
- 复杂胜于凌乱。

此外，Python 还同时支持面向过程和面向对象的编程。Python 的可扩展性也很强，可以用 C/C++ 为 Python 编写扩展，Python 也可以嵌入到 C/C++ 编写的程序之中，因此，Python 语言也常被称为“胶水语言”。

Python 语言的这些优点就是选择它作为学习和使用的一门编程语言的理由，也是笔者要向读者介绍它的原因。

1.3 安装 Python

Python 语言是解释型的高级程序设计语言，要使用它就必须安装 Python 的编译系统（解释器）。

如前文所述，Python 可以运行在最常见的 Windows、Linux、Mac 等系统的计算机中，当然也包括以上系统的 32 位版本和 64 位版本。

如果你的系统不在这三个之中，应该可以找到能在你所使用系统的对应 Python 语言版本进行安装。

Python 语言目前具有两种版本：Python 2.x（目前最新版本是 2.7.9）和 Python 3.x 版本（目前最新版本是 3.6.2），它们的语法和标准库是有差别的。用户可以根据自己的需要选择进行下载和安装。本书讲述的是 Python 3.x 版本语法和标准库，这里建议下载 Python 3.x 版本，以便于学习和运行本书中的实例。本节介绍的安装 Python 方法，安装的也是 Python 3.x 版本。

1.3.1 在 Windows 下安装 Python

下面以稳定版 3.6.2 为例来介绍，在 Windows 系统下安装 Python 的步骤如下所示。

(1) 到 <https://www.python.org/> 网站去下载 Python3 版本的 Windows 的安装程序，其 64 位版的下载地址为：<https://www.python.org/ftp/python/3.6.2/python-3.6.2.amd64.exe>，32 位版的下载地址为：<https://www.python.org/ftp/python/3.6.2/python-3.6.2.exe>。

(2) 双击下载的文件，开始安装 Python 3.6.2，基本上一路单击“Next”按钮就可以了，当然在安装过程中，也可以自定义一些安装选项。

(3) 启动安装，如图 1.3 所示，Install Now 意为直接以默认选项安装 Python，一般使用默认的第一项即可。Customize installation 意为定制安装。

(4) 自定义安装项目选项，如图 1.4 所示，将这几个选项都勾选上（自动安装 pip 等软件包），单击“Next”按钮。

(5) 选择安装选项，如图 1.5 所示，你可以进一步选择自定义安装项目，如果没有特殊要求，可使用默认的目录，否则应输入另一个目录路径进行安装，一般来说使用默认选择，单击“Install”按钮。

此后，会在安装窗口中出现复制安装文件的进度条，等待窗口中显示“Finish”按钮时，单击它就完成了 Python 的安装。



图 1.3 启动安装

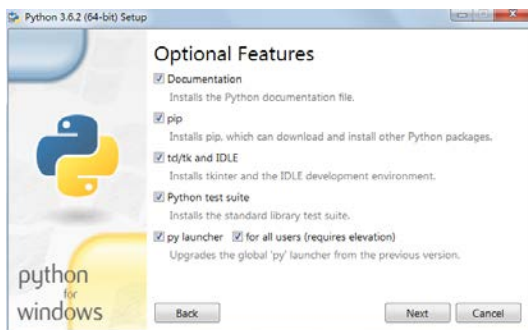


图 1.4 自定义安装项目选项

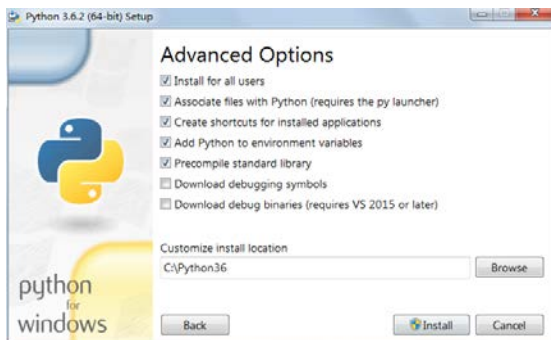


图 1.5 选择安装选项

此外，在 Windows 系统中，为了在命令行（CMD）模式下可以使用 Python 命令，如果系统提示找不到这个命令，需要添加或修改环境变量，具体操作如下：



(1) 右击“我的电脑”，选择“属性”菜单，如图 1.6 所示。

(2) 单击窗口左侧的“高级系统设置”链接，在弹出的“系统属性”窗口（如图 1.7 所示）中，选择“高级”选项卡，单击“环境变量”按钮。

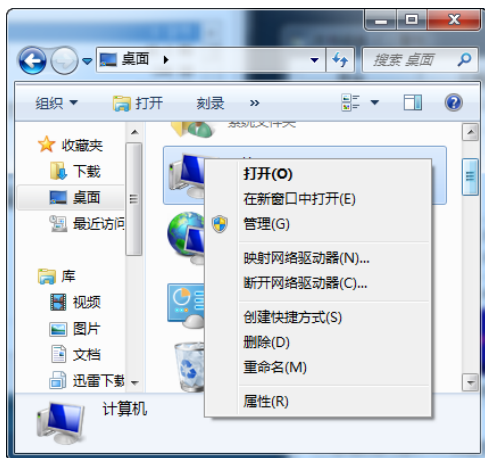


图 1.6 资源管理器窗口

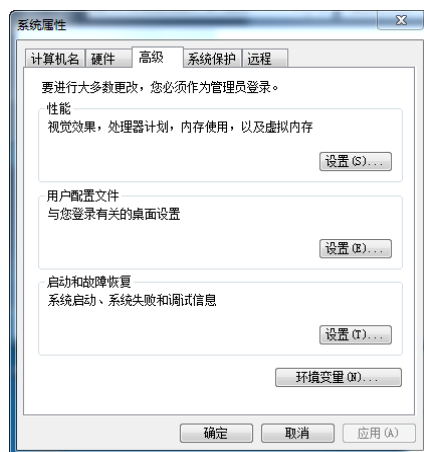


图 1.7 系统属性窗口

(3) 在弹出的“环境变量”窗口（如图 1.8 所示）中添加 Lenovo 的用户变量。在“系统变量”中选中变量“Path”，然后单击“编辑”按钮，在弹出的“编辑系统变量”对话框（如图 1.9 所示）的“变量值”框中填写你前面自定义的安装目录。

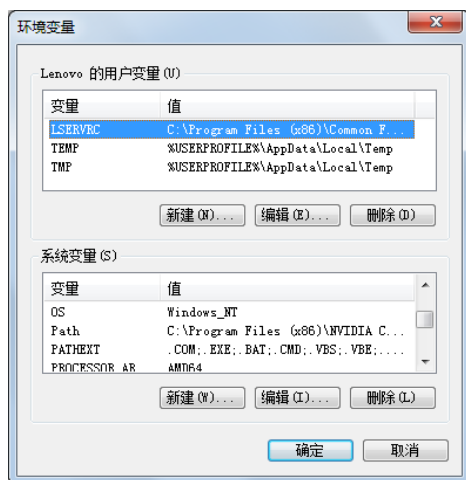


图 1.8 环境变量窗口

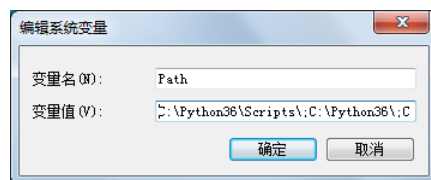


图 1.9 新建用户变量对话框

(4) 最后依次单击“确定”按钮，直到关闭了所有对话框。

1.3.2 在 Linux 下安装 Python

在 Linux 系统下安装 Python 的方法如下。

一般 Linux 系统的桌面版会自动安装有 Python 2.x，本书介绍 Python 3.x，所以需要读者安装安装 Python 3.x。在 Linux 下安装 Python 可以有两种方法：

- 在 Linux 中直接使用其安装命令，通过网络来安装 Python 3.x；
- 直接到 Python 官网下载源码并编译安装。

在可以使用 `yum` 安装命令的 Linux 系统, 如 Cent OS、Fedora 等操作系统中, 用以下命令根据提示安装:

```
yum install python3
```

在可以使用 `apt-get` 安装命令的 Linux 系统, 如 Debian 等操作系统中, 用以下命令根据提示安装:

```
apt-get install python3
```

如果网络不稳定或者安装不了 Python 3.x, 也可以下载其源码, 进行编译安装, 基本步骤为:

(1) 到下列网址下载源码:

<https://www.python.org/ftp/python/3.6.2/Python-3.6.2.tgz>

或

<https://www.python.org/ftp/python/3.6.2/Python-3.6.2.tar.xz>

(2) 在终端命令模式下用以下命令解压下载的压缩包:

```
tar -xzf Python-3.6.2.tar.xz
```

(3) 在终端命令模式下进入解压后的子目录并依次使用以下命令进行安装 (如果提示安装错误或缺少某个依赖库, 请先行安装, 再重新运行以下命令):

```
./configure  
make install  
make
```



注意 以上安装都需要 root 用户权限完成

顺便说一下, Python 语言有很多的第三方库, 在用户进行项目开发时是可以自由使用的, 如果需要的话可以下载并安装。

下载安装第三方库一般有以下几种方法。

(1) 最简单的方法是使用 `pip` 来进行安装, 其基本命令如下 (Windows 要在命令行下, Linux 在终端下):

```
pip install libname
```

`libname` 是要下载安装的第三方库的名称, 而且它还会自动下载和安装其依赖的第三方库。

(2) 自行下载第三方库的压缩包并解压缩, 然后在命令提示符或终端下进入对应的目录, 执行以下命令:

```
python setup.py install
```



注意 如果你的系统同时安装了 Python 2.x 和 Python 3.x, 为 Python 3.x 安装第三方库时应使用的命令为 `python3 setup.py install`

这种安装方法有一个缺点是有时不能自动下载安装依赖库, 你只能通过在安装过程中提供的信息了解依赖的第三方库或查看要安装第三方库的文档了解其依赖库, 并先行下载安装。

此外, 在 Linux 下, 还可以使用系统提供的安装命令 (如 `yum`) 来安装它。



注意 在 Linux 下, 以下安装命令前必须要加 `sudo` 或先 `su` 进入可以安装软件的用户权限中才可以安装。

寻找和下载第三方库最常用的就是网站 <https://pypi.python.org>, 还可以在此网站中搜索想要的第三方库, 当然你还可以通过搜索引擎来搜索相关的第三方库来下载安装。



1.4 选择开发工具

开发工具对于一个程序开发者来说主要是习惯的问题，各种开发工具都具有不同的特点，你可以自由选择。Python 语言的开发工具也很多，有收费的，也有免费的，甚至还可以使用系统附带的文本编辑工具，比如在 Windows 下的记事本、Linux 下的 vi、gedit 等。



1.4.1 Python 自带开发工具：IDLE

Python 自带开发工具——IDLE，它是应用 Python 第三方库的图形接口库 tkinter 开发的一个图形界面的开发工具，其基本特点如下：

- 跨平台使用，包括 Windows、Linux、Unix 和 OS X；
- 智能代码缩进；
- 自动代码着色；
- 自动代码提示；
- 可以调试代码，包括设置断点、单步执行等；
- 智能化菜单。

在 Windows 下安装 Python 时，IDLE 会自动安装，在开始菜单的 Python 3.x 子菜单中就可以找到它；在 Linux 下可以使用 yum 或 apt-get 单独进行安装。在 Windows 下其界面如图 1.10 所示，标题栏与普通的 Windows 应用程序相同，而其中所写的代码是被自动着色的。此外，其常用的快捷键及其意义如表 1.1 所示。

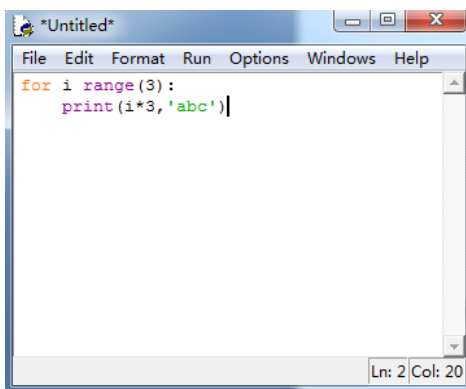


图 1.10 IDLE 窗口界面

表 1.1 IDLE 常用的快捷键及其意义

快捷键	意 义
Ctrl+]	缩进代码
Ctrl+[取消缩进
Alt+3	注释代码
Alt+4	去除注释
F5	运行代码
Ctrl+Z	撤销一步

1.4.2 文本编辑器：Emacs

世界上所讨论最多的文本编辑器就是 Vim 和 Emacs，而且关于是否选用 Vim 还是 Emacs 的争论从来就没有停止过。

Emacs 被设计成“无所不能”的，号称是世界上最强大的文本编辑器。使用 Emacs 就像使用 Windows 的记事本一样，但 Emacs 比 Windows 的记事本的功能要强大得多。

可以从 <http://ftp.gnu.org/gnu/emacs/windows/> 网站下载编译好的 Windows 安装程序，然后在 Windows 下进行安装。Emacs 安装程序是一个自解压的压缩文件，只需选择解压目录进行解压。解压完成后，运行 Emacs 所在目录下“bin”目录中的“runemacs.exe”文件，即可启动 Emacs 程序，其工作界面如图 1.11 所示。

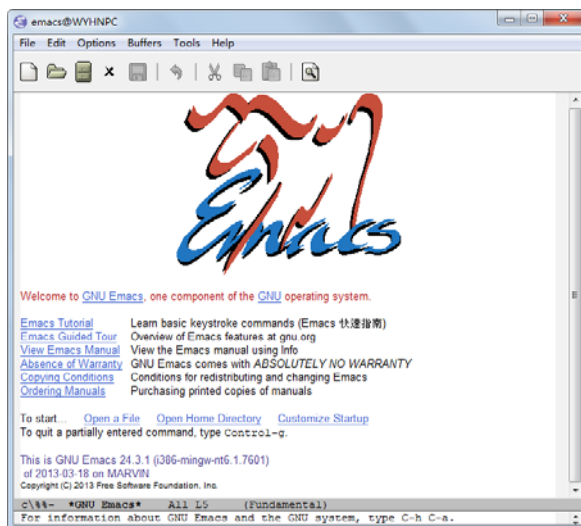


图 1.11 Emacs 文本编辑器

Emacs 中常用的命令及其意义如表 1.2 所示。

表 1.2 Emacs 常用的命令及其意义

命 令	意 义
C-v	向后翻一页
M-v	向前翻一页
C-l	将当前行居中
C-f	向前移动一个字符
M-f	向前移动一个单词
C-b	向后移动一个字符
M-b	向后移动一个单词
C-n	向下移动一行
C-p	向上移动一行
C-a	移至当前行的第一个字符
M-a	移至当前所在句子的第一个字符
C-e	移至当前行的最后一个字符
C-p	移至当前所在句子的最后一个字符



续表

命 令	意 义
M-<	移动到当前窗口的第一个字符
M->	移动到当前窗口的最后一个字符
C-x C-c	永久离开 Emacs
C-x C-f	读取文件到 Emacs
C-x r	只读的方式打开一个文件
C-x C-q	清除一个窗口的只读属性
C-x C-s	保存文件到磁盘
C-x s	保存所有文件
C-x i	插入其他文件的内容到当前缓冲
C-x C-v	用将要读取的文件替换当前文件
C-x C-w	将当前缓冲写入指定的文件
C-s	向前查找
C-r	向后查找
C-M-s	规则表达式查找
C-M-r	反向规则表达式查找
M-p	选择前一个查找字符串
M-n	选择下一个查找字符串
C-d	向前删除字符
M-d	向前删除到字首
M-DEL	向后删除到字尾
M-0 C-k	向前删除到行首
C-k	向后删除到行尾
C-x DEL	向前删除到句首
M-k	向后删除到句尾
M-- C-M-k	向前删除到表达式首部
C-M-k	向后删除到表达式尾部
C-x r r	复制一个矩形到寄存器
C-x r k	删除矩形
C-x r y	插入刚刚删除的矩形
C-x r o	打开一个矩形，将文本移动至右边
C-x r c	清空矩形
C-x r t	为矩形中每一行加上一个字符串前缀
C-x r i r	从 r 缓冲区内插入一个矩形
C-x 1	删除所有其他窗口
C-x 2	上下分割当前窗口
C-x 3	左右分割当前窗口
C-x 0	删除当前窗口
C-M-v	滚动其他窗口
C-x o	切换光标到另一个窗口
C-x ^	增加窗口高度

续表

命 令	意 义
C-x {	减小窗口宽度
C-x }	增加窗口宽度

需要说明的是,上面列出的命令中的“C”代表按下 Ctrl 键,“M”代表按下 Alt 键。例如命令“C-v”是指按着 Ctrl 键不放然后再按“v”键。而命令“C-x C-f”则指按住 Ctrl 键不放,先按下“x”键,然后松开“x”键再按“f”键。而“M->”命令则指先按住 Alt 键不放,然后再按住 Shift 键不放,然后再按“.”,这是因为按住 Shift 键不放然后按“.”才是“>”。

1.4.3 Python 开发工具: PythonWin

前面所提到的 Vim 和 Emacs 两种文本编辑器各有特色,但上手较难。尤其是在操作 Emacs 的时候需要使用大量的快捷键,初学者很难记住。不过,一旦习惯,并且深入学习 Vim 和 Emacs 后,会发现它们确实非常强大。

在 Windows 下还有一个比较好用的 Python 程序编辑器——PythonWin 所附带的编辑器。PythonWin 是 Python 在 Windows 下的扩展包,使用 PythonWin 可以让 Python 使用 Windows 系统的特性。

1. PythonWin 安装

在 Windows 下使用 Python 最好安装 PythonWin 扩展包。在 PythonWin 中提供了 Win32API 函数的封装,以及 MFC 类库的封装,通过 PythonWin 的相关模块可以在 Python 中直接调用 Windows 的 API 函数。PythonWin 的安装步骤如下。

(1) 从 PythonWin 官方网站 [http://sourceforge.net/projects/pywin32/files/pywin32/Build 219/](http://sourceforge.net/projects/pywin32/files/pywin32/Build%20219/) 下载 PythonWin 的安装程序 pywin32-219.win-amd64-py3.4.exe (注意文件名中的 py3.4 表示适用于 Python 3.4 版)。

(2) 双击运行安装程序后如图 1.12 所示。

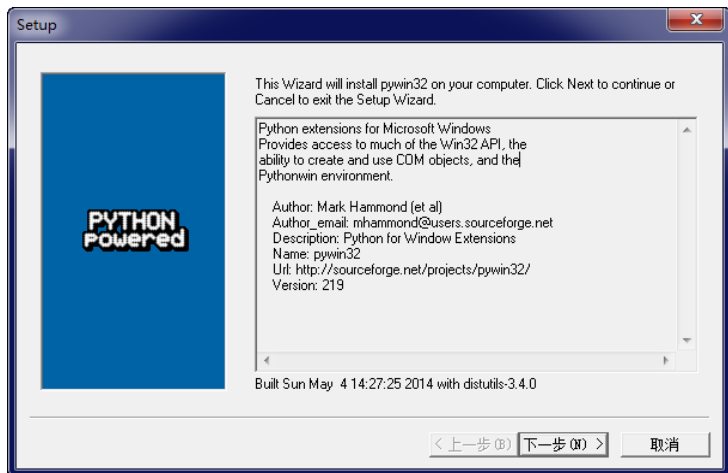


图 1.12 PythonWin 安装程序

(3) 单击“下一步”按钮,安装程序将自动搜索 Python 的安装路径,如图 1.13 所示。如果未能找到 Python 的安装路径,则需要检查 Python 的版本是否与 PythonWin 的版本相对应,或者重新安装 Python。

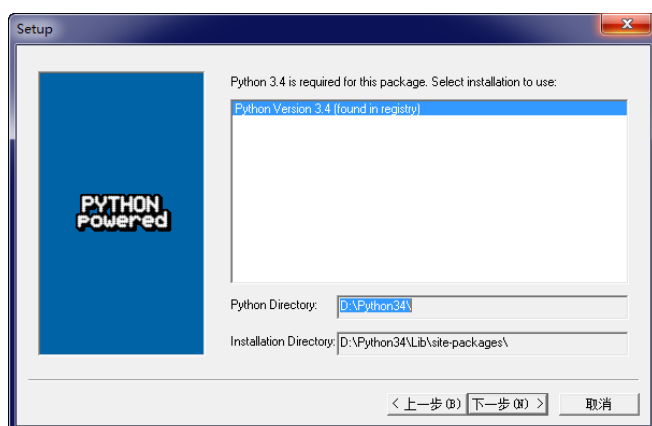


图 1.13 Python 安装路径

(4) 单击“下一步”按钮，进入确认安装界面，如图 1.14 所示。

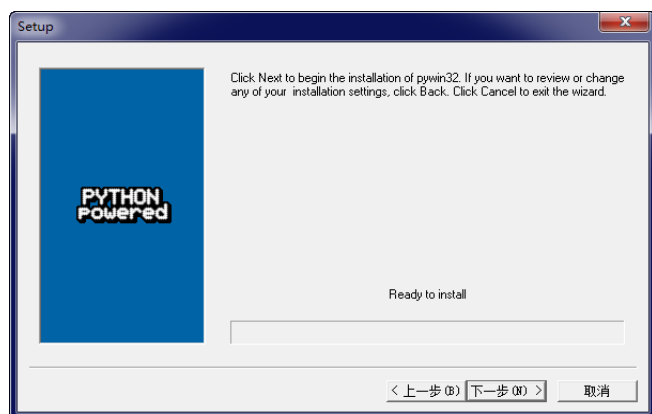


图 1.14 确认安装

(5) 单击“下一步”按钮，PythonWin 的安装程序将开始复制文件。当文件复制完成后，将出现如图 1.15 所示的界面。单击“完成”按钮，即可完成 PythonWin 的安装。

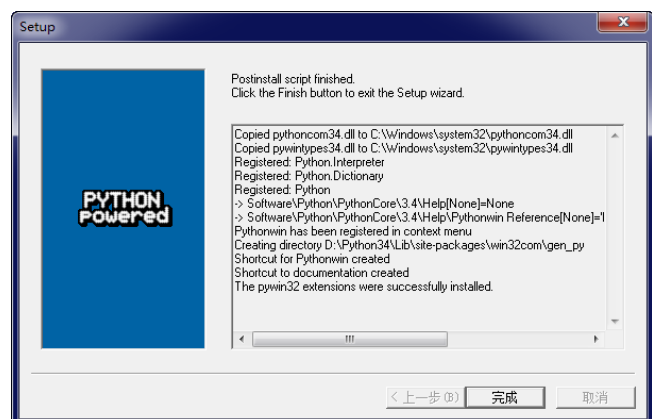


图 1.15 完成安装

2. PythonWin 编辑器简介

PythonWin 提供了一个简单易用的编辑器。在 PythonWin 中不仅可以交互式地运行 Python 命令，还可以编写 Python 程序。单击“开始”→“所有程序”→“Python 3.4”→“PythonWin”命令，将打开 PythonWin 集成环境，如图 1.16 所示。PythonWin 将自动打开 Python 的交互式命令行窗口。单击“File”→“New”命令，可以新建 Python 程序。

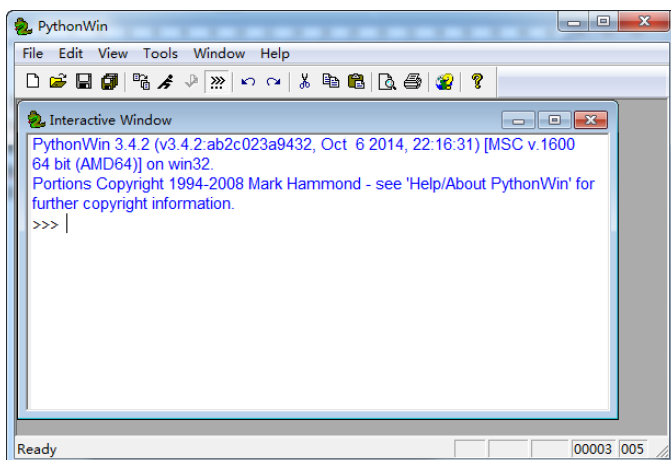


图 1.16 PythonWin 集成环境

PythonWin 中也提供了自动补全的功能，例如当导入模块后，在模块名后输入“.”以后，PythonWin 将弹出一个列表窗口，使用方向键上和下可以选择列表中的项目，按下 Tab 键可以补全，如图 1.17 所示。

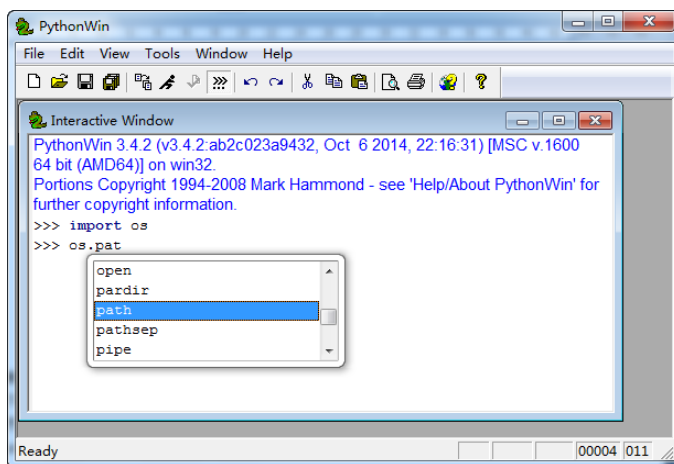


图 1.17 模块函数自动补全

如果所输入的变量已经输入过，则可以按“Alt+/”键来自动补全。另外，在 Python 的交互式命令行下可以按 Ctrl 键加向上的方向键（或者向下的方向键），以此来不重复地输入前面所输入的命令。

当然，除了以上介绍的开发工具之外还有 Sublime Text、BlackAdder、Wing IDE、Komodo、Boa Constructor、PyDev、Eric3、DrPython 和 SciTE 等，你可以根据个人习惯和爱好自由选择。



1.5 编辑和运行 Python 程序

在 Windows 下有多种方式可以运行 Python 程序，也可以直接在 Python 的交互式命令行下一句一句地编写运行 Python 程序。当代码很多的时候，则应该在文本编辑器中将代码编辑好，然后再运行。

1.5.1 你好，Python

使用 Python 编写最简单的程序仅需要一行代码，如下所示。

```
print('你好,Python!')
```

打开文本编辑器（或本章前面介绍的任意一款开发工具软件），输入上述代码（如图 1.18 是在 IDLE 中输入以上代码的界面），然后将其保存为“hello.py”。

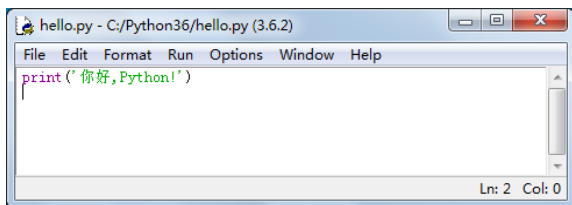


图 1.18 在 IDLE 中编写第 1 个 Python 程序



注意

输入程序代码时，只有当需要输入中文时才转换为中文输入法，一般情况下就使用英文输入法，防止字符录入错误而导致程序不能运行。

1.5.2 运行程序文件

对于 Python 程序文件，比如 1.5.1 小节所保存的 hello.py 程序文件运行的方法也有多种：

(1) 按 F5 快捷键运行程序或使用 Run 菜单下的 Run Module 命令运行，运行效果如图 1.19 所示。

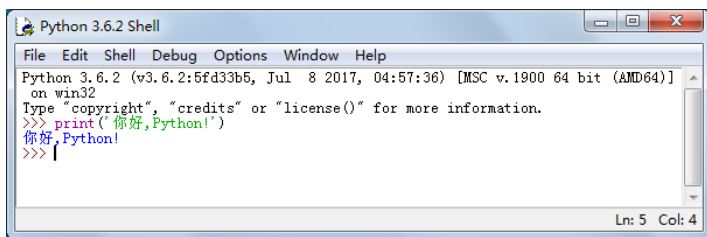


图 1.19 在 GUI Python Shell 中运行并显示出结果

(2) 在 Windows 下可以通过直接双击来运行。如果双击运行上面编写的程序文件“hello.py”，可以看到一个命令行窗口出现，然后又关闭，由于很快，看不到输出内容，因为程序运行结束后立即退出了。

(3) 为了能看到程序的输入内容，可以按以下步骤进行操作。

①单击“开始”菜单，在“搜索程序和文件”文本框中输入“cmd”，并按 Enter 键，打开 Windows 的命令行窗口。

②直接输入 hello.py 文件的绝对路径及文件名或“python d:\lx\hello.py”，再按 Enter 键运行程序。也可以使用 cd 命令，进入“hello.py”所在的目录，如“D:\lx”，在命令行提示符下输

入“hello.py”或者“python hello.py”，然后按 Enter 键即可运行“Hello.py”程序，使用三种方式运行了 hello.py 程序文件，效果如图 1.20 所示。

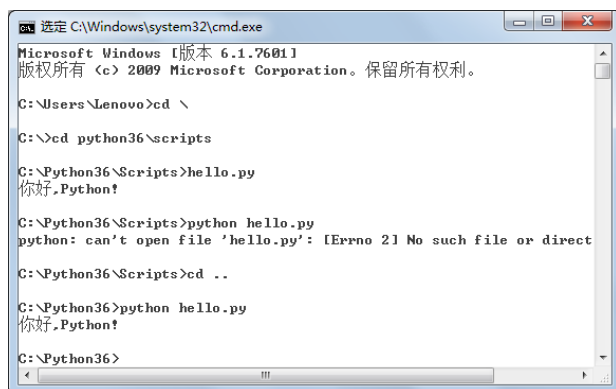


图 1.20 在命令提示符下运行 Python 程序

(4) 在 Linux 系统中，在 Terminal 终端的命令提示符下可以使用 `python hello.py` 命令来运行 Python 程序。

1.5.3 交互式运行 Python

Python 还提供了交互式命令行操作环境，可以一边输入程序，一边运行程序。

在 Windows 下通过单击“开始”→“所有程序”→“Python 3.6”→“IDLE (Python 3.6 GUI - 64 bit)”菜单命令，可以打开 Python 的交互式命令行，在命令行中输入“`print('你好,Python!')`”，然后按 Enter 键，运行的结果如图 1.21 所示。

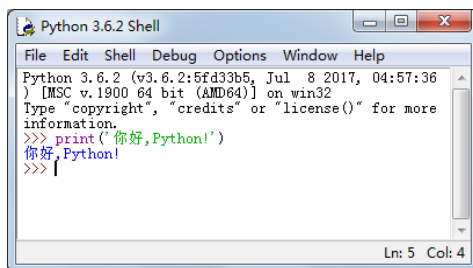


图 1.21 交互式运行 Python 及运行结果

在 Windows 下还可以使用另外一种方式的交互式运行 Python 的环境，其启动方法如下：

(1) 单击“开始”菜单，在“搜索程序和文件”文本框中输入“cmd”，并按 Enter 键，打开 Windows 的命令行窗口。

(2) 输入“python”后按 Enter 键就进入了交互式运行环境，其效果如图 1.22 所示。

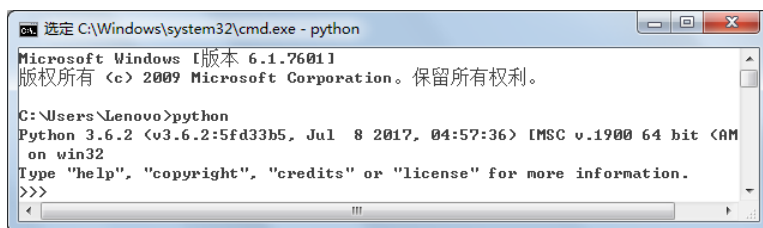


图 1.22 命令提示符界面下交互式运行 Python



在 Linux 中也可以通过在 Terminal 终端的命令提示符下运行命令 `python` 来启动 Python 的交互式运行环境来边输入程序边运行程序。

1.6 小结

本章主要介绍了编程的基本常识（包括计机构成基础知识、程序语言概况及调试程序），Python 语言的优势，在不同的系统下安装 Python、Python 程序的开发工具，最后介绍了如何利用 Python 开发工具来输入程序、运行程序及在交互式环境的启动与使用，还让读者体验了一个最简单的 Python 程序，为继续学习 Python 语言打下扎实的基础。

1.7 本章习题

一、选择题

1. 计算机中信息处理和信息存储使用（ ）。

- A. 二进制代码
- B. 十进制代码
- C. 十六进制代码
- D. ASCII 码

【解析】由计算机的物理部件决定，计算机中任何信息都是用二进制代码表示。答案为 A。

2. 一个完整的计算机系统应包括（ ）。

- A. 主机和外设
- B. CPU 和内存
- C. 硬件和软件
- D. 软件和主机

【解析】如本章内容所述，一个完整的计算机系统由硬件和软件组成。答案为 C。

3. Python 语言源程序的执行方式是（ ）。

- A. 编译执行
- B. 解释执行
- C. 直接执行
- D. 边编译边执行

【解析】高级语言源程序的执行方式只有两种，即编译和解释；而 Python 源程序为解释执行方式，答案为 B。

4. Python 语言源程序的语句块的标志是（ ）。

- A. 分号
- B. 逗号
- C. 缩进
- D. \

【解析】用缩进来标志语句块，是 Python 语言的特点。答案为 C。

二、实验题

1. 请给你的计算机下载并安装 Python 语言和一种 Python 语言的开发工具。

2. 请使用 Python 开发工具，分别在交互式环境下、IDLE、命令提示符（或 shell 中）写入以下一段代码，并尝试运行以下生成验证码的代码（生成一串随机的六个包含字母和数字的字符）：

```
import string,random                                #引入相关标准库

capta = ''
words = ''.join((string.ascii_letters,string.digits)) #生成大小写字母和数字串
for i in range(6):
    capta += random.choice(words)                    #随机选择一个字母或数字
print(capta)                                         #打印生成的验证码字符串
```

第 2 章 Python 起步

任何一门语言（包括自然语言和编程语言）都有一定的语法和语义规范，要学习一门语言就必须先学习其最基本的语法和语义规范。

Python 的语法简单，学习和掌握起来都很容易；在 Python 语言学习的初始阶段，本章先对其做整体介绍，以期望读者能先从整体上把握编写程序的基本结构及写法，避免在上机实践时出现一些低级错误而又解决不了。

本章的主要内容：

- Python 语法基础；
- Python 输入输出；
- 在交互环境下用 Python 计算。

2.1 Python 语法基础

用 Python 编写程序，就要按照它的代码组织形式、语法和语义来书写程序。因此，首先要学习代码组织方法、注释等。



2.1.1 缩进分层

如果你学习过其他的高级程序设计语言，就会了解到，为使程序代码结构清晰，必须要进行缩进，即使写在同一行内也是正确无误的。比如以下的一段 C 语言程序：

```
void main()
{
    int t,i;
    for(i=2;i<=100;i++)
    {
        t=1;
        for(t=2;t<i;t++)
        {
            if(i%t==0)
            {
                t=0;
                break;
            }
            if(t==1) printf("%d",i);
        }
    }
}
```

如果不考虑缩进，完全可以写在同一行内（同一行内能写下的话），如下：

```
void main(){ int t,i; for(i=2;i<=100;i++) {t=1;
for(t=2;t<i;t++){if(i%t==0){ t=0; break; }if(t==1) printf("%d",i); }}
```

但要让你去看懂这一行式代码的话，简直是“难于上青天”！

通过以上一行式代码可以看出，代码是通过括号、分号、大括号等进行语句或语句块分隔的，而 Python 程序要求的代码最好是全部使用缩进来分层（块）。代码缩进一般用在函数定义、类的定义以及一些控制语句中。一般来说，行尾的“:”表示下一行代码缩进的开始，以下的一段复杂的代码中就有在分支语句中使用缩进，即使没有使用括号、分号、大括号等进行语句（块）的分隔，通过缩进分层的结构也非常清晰：



```
if a > b:
    if a == 1:           # 代码缩进
        print(a)         # 代码缩进（缩进嵌套）
    else:
        if a == 0:
            print(a)
        else:
            pass
elif a == b:
    print(a,b)
else:
    print(b)
```

以上代码中虽然包含大量的嵌套结构，但会通过缩进使你很容易分清哪个 `if` 与 `else` 或 `elif` 相配的条件判断，同时也让程序员减少编程时去大量录入只有分隔语句或语句块作用的括号、分号、大括号等，从而提高程序员的开发效率。而编写 Python 时，可以使用一些具有自动缩进的编辑器来减少输入。

要注意的是，处于同一级的代码缩进量要保持一致，并且缩进的符号（Tab 键、空格等）也应保持一致，这样才能保证嵌套正确。否则，这种不一致的缩进会导致错误，甚至程序不能运行。例如以下代码运行时就会提示语法错误而且不能正常运行：

```
if a > b:
    print(a)
    print('>')           #此处代码少缩进了两个空格
    print(b)
```

Python 的编程规范指出：缩进最好采用空格的形式，每一层向右缩进 4 个空格。一般不建议使用 Tab 键进行缩进。

此外，有不少开发工具是可以定义按一次 Tab 键产生 4 个空格的缩进的；还有一些工具可以自动地进行代码缩进，这都给程序员带来很大的方便。

2.1.2 代码注释

代码注释是程序中不可少的部分，不仅可以作为程序员之间交流的主要途径，还可以方便作者自己以后阅读代码与维护代码。

Python 中的注释有两种形式：

(1) 单行注释，以“#”字符开始，同一行中其后的所有内容都视为注释，不论是什么都不会执行。

(2) 多行注释，用三个单引号“'''”或三个双引号“"""”将注释的内容包围起来。

以下代码同时包含有这两种注释：

```
'''
该程序段的功能是：
根据变量 x 值输出+/-
'''
if a >= 0:
    print('+')          #大于等于 0 输出+
else:
    print('-')          #不大于 0 输出一
# print(a)             #此行为注释语句不会被执行
```

2.1.3 断行

前文说过，Python 代码中不需要用分号来分隔语句，直接将一条语句写在一行之内。但是 Python 中其实也可以将两条语句字书写在同一行而中间用分号隔开的。

还有一种情况就是，如果缩进语句块中只有一条语句，也可以直接写在“:”之后的，以下代码也是正确的：

```
print('+');print('-')    #用“;”分隔的写在同一行的两条语句
if a>0:print('+')       #缩进的语句只有一条而写在同一行内
else:print('-')         #缩进的语句只有一条而写在同一行内
```

当然，如果 Python 代码中一条语句过长或为了结构清晰而不能或不方便写入同一行内，这时可以使用“\”将该行后的内容写入下一行，而“\”后则不允许有任何内容。以下代码使用“\”将同一语句写入三行是正确的：

```
print('I am a teacher',\
      first_name,\
      last_name)
```

以上代码，其实去掉“\”也是正确，因为 Python 语言也规定了圆括号包围的部分是可以写在不同行的。



注意 在使用“\”进行续行的情况下，“\”之后是不能放任何字符或单行注释的。

2.2 Python 输入/输出



对于所有的程序，输入和输出是用户与程序进行交互的主要途径，通过输入程序能够获取程序运行所需的原始数据，通过输出程序能够将数据的处理结果输出，让用户了解运行结果。

2.2.1 接收键盘/输入

Python 程序如果需要输入，就必须调用其 `input()` 函数，基本形式如下：

```
input([prompt])
```

其中的参数是可选的，即可以使用，也可以不使用。参数是用来提供用户输入的提示信息字符串。当用户输入程序所需要的数据时，就会以字符串的形式返回。



注意 用户输入的数据全部以字符串形式返回，如果需要输入数值，则必须进行类型转换。

【实例 2-1】 以下代码演示用户输入姓名，并将姓名以字符串形式返回并接收在 `name` 变量名中，以后可以使用 `name` 名字来引用它：

```
name = input('Please input your name:')
```

【代码说明】 代码中 `input()` 是函数调用的格式，这个函数是 Python 中的内建函数，直接调用就可以了。函数中的“Please input your name:”可选参数的作用是：当程序要求用户输入信息时，会显示一条提示信息“Please input your name:”，这样用户就会知道将要输入的是什么数据，否则用户看不到相关提示，可能认为程序正在运行，而在等待运行结果，甚至会使用户不知所措，这也是编程时所需要的友好用户界面。

【运行效果】 输入的姓名可以使用 `name` 来引用它，如图 2.1 所示，在交互式环境下运行此句代码，第二行立即显示提示信息“Please input your name:”，之后等待用户的输入。当用户输入“Bob”并按下“Enter”键时，程序就接收了用户的输入。之后，使用 `name` 变量名，就会显示变量所引用的对象——用户输入的姓名。



```
>>> name = input('Please input your name:')
Please input your name:Bob
>>> name
'Bob'
>>> |
```

图 2.1 输入函数的运行结果



图 2.1 中位于“>>>”之后的是用户输入的 Python 语句，按“Enter”后会立即执行。没有“>>>”的行都是 Python 语句运行时的输出信息。

2.2.2 显示处理结果

Python 程序如果需要输出，就必须调用其 `print()` 函数，基本形式如下：

```
print(value, ..., sep=' ', end='\n') #此处只说明了部分参数
```

其中的参数的意义如下：

- `value` 是用户要输出的信息，后面的省略号表示可以有多个要输出的信息；
- `sep` 是多个要输出信息之间的分隔符，其默认值为一个空格；
- `end` 是一个 `print()` 函数中所有要输出信息之后添加的符号，默认值为换行符。

看起来是不是很眼熟呢，还记得在第 1 章中的一行代码的程序吗？只有一句：

```
print('你好,Python!')
```

其中的参数只有一个要输出的信息：‘你好,Python!’

【实例 2-2】演示了输出函数的示例，代码如下：

```
print('a','b','c')
print('a','b','c',sep=',')           #将默认分隔符修改为 ‘,’
print('a','b','c',end=';')          #将默认结束符号修改为 ‘;’
print('a','b','c')
```

【代码说明】此代码中使用了四条语句，调用了四次 `print()` 函数。其中第二条语句将分隔符改为 ‘,’，第三条语句将结束符号改为 ‘;’。

【运行效果】如图 2.2 所示，第一行为默认的输出，数据之间以空格分开，结束后添加了一个换行符；第二行输出的数据项之间以 ‘,’ 分开；第三行输出结束后添加了 ‘;’，所以和第四条语句的输出放在了同一行中。

```
>>>
a b c
a,b,c
a b c;a b c
>>>
```

图 2.2 输出函数演示

2.3 用 Python 计算

在 Python 的交互式环境下，可以将其作为一个表达式计算器使用。你只要输入需要计算机的表达式，Python 就会很快地计算出结果，非常方便。Windows 系统自带的计算器还不能计算表达式呢！此外，Python 还提供了功能丰富的数学计算函数，你



都可以在交互式环境下自由使用。

2.3.1 算式与代数式运算

由于 Python 是解释方式执行的高级程序设计语言，在交互式命令环境下可以输入算式进行计算，Python 会立即算出结果，并在交互式环境下显示。更为高级的是还可以预定义变量的值，并计算代数式的值。示例运算的算式及结果如下：

```
>>> 3*5/2+4*2          #基本四则混合算术运算式
15.5
>>> 27*(4-2)/(2+4*(3+1))  #用括号改变优先级，并能嵌套使用
3.0
>>> 2**3                #求 2 的 3 次方
8
>>> x = 3                #定义变量 x 的值为 3
>>> 2*x**2-x+4           #计算代数式的值
19
>>> y = 5                #定义变量 y 的值为 5
>>> x**2+y**2            #计算代数式的值
34
```



注意 算式之中的乘号不可以省略。

接下来你就可以自由发挥，计算你想要的任何算式或代数式了吧？其实 Python 标准库中还内置了一个 math 模块，提供了丰富的数学函数，其中的常用数学函数及其功能如表 2.1 所示。

表 2.1 math模块常用的函数及其功能

函 数	功 能
sin(x)	求 x 的正弦
cos(x)	求 x 的余弦
asin(x)	求 x 的反正弦
acos(x)	求 x 的反余弦
tan(x)	求 x 的正切
atan(x)	求 x 的余切、反正切
hypot(x, y)	求直角三角形的斜边长度
fmod(x,y)	求 x/y 的余数
ceil(x)	取不小于 x 的最小整数
floor(x)	取不大于 x 的最大整数
fabs(x)	求绝对值
exp(x)	求 e 的 x 次幂
pow(x,y)	求 x 的 y 次幂
log10(x)	求 x 以 10 为底的对数
sqrt(x)	求 x 的平方根
pi	π 的值

因为它不是 Python 的内建函数，所以在使用前要用以下语句进行导入：

```
import math
```

以下是应用 math 模块中函数在交互式环境下进行计算的一些示例：



```
>>> import math                #导入 math 模块
>>> x = 2                      #创建变量并赋值
>>> math.sin(x)                #求正弦值
0.9092974268256817
>>> math.hypot(3,4)            #求 3, 4 为直角边的三角形斜边长
5.0
>>> math.sqrt(x)               #求平方根
1.4142135623730951
>>> math.asin(math.sqrt(2)/2)  #嵌套调用, 求 2 的平方根除以 2 的反正弦值
0.7853981633974484
>>> math.pow(2,16)             #求 2 的 16 次方
65536.0
```



注意

使用 math 模块中的函数时, 其前面应加上 “math.”, 表示调用 math 模块中的函数。

2.3.2 惊奇

如图 2.3 所示, 使用交互式环境计算 199 的 99 次方, 居然毫无压力而又完整地计算出了结果!

```
>>> 199**99
38588203890351162888311418102842483332854160575348157934331975488859361121287141
83960276629953622210527651402082986279843678863751946778417112798453219771984728
9403643242818934140224374438980617671424886485298541220033299797979
```

图 2.3 交互式环境中计算 199 的 99 次方

在 Python 中, 直接提供了对大整数的支持, 用户可以直接使用。

而在大多数程序设计语言中, 保存整数的变量都有一个值的区域, 当超过该区域后, 就无法保存更大的数。例如, 在 C 语言中, int 类型的变量使用 4 个字节来保存值, 保存数据的范围为 -2 147 483 648 ~ 2 147 483 647。如果计算结果为更大的值, 则无法保存, 即使使用 long 类型, 仍然有一个范围限制。这样, 就无法进行如阶乘、大的幂运算等, 要保存这些结果值很大的数据, 就需要另外编写大整数处理程序。

如图 2.4 所示, 第一行是在交互式环境中计算了 9 个 0.1 相加, 看到输出结果为 0.8999……, 咦, 结果应该是 0.9 啊? 这其实就是计算机表示数的方法与我们使用的表示方法不同: 我们使用的是十进制数, 而计算机使用的是二进制数, 0.1 是无法精确地转换为二进制数的, 所以就产生了误差。无论哪种编程语言都存在这个问题, 只不过有的编译或运行时在内部进行了处理, 你看不到其“错误”的结果。

```
>>> 0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1
0.8999999999999999
>>> 0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1==0.9
False
>>> 0.9-(0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1) < 10**-9
True
```

图 2.4 小数的计算与比较

那么, 如果要比较 0.9 和 9 个 0.1 相加的大小呢? 在图 2.4 中的第三行, 两个等于号 “==” 代表判断它们是否相等, 其后一行显示的是 False, 显示是不相等的。明明它们应该是相等的! 在程序设计中要如何处理这个问题呢? 方法是计算它们的差是不是小于一个极小的数, 在如图 2.4 所示的第五行中, 计算它们的差是否小于 10 的 -9 次方, 在其后显示的结果中为 True, 显然这就符合要求。



要注意 要比较两个浮点数是否相等，则应看它们差值是否小于一个极小的数。

2.4 小结

本章主要讲述了 Python 程序的代码组织形式（包括缩进分层、注释代码与断行）、程序与用户的交互（输入、输出）函数的应用。最后讲述了 Python 语言对大整数的支持，以及如何处理浮点数的运算与比较。

通过本章学习，应掌握：

- 缩进分层、代码注释与断行；
- input()函数；
- print()函数；
- 浮点数比较。

2.5 本章习题

一、选择题

1. Python 语言的源程序缩进不可以使用（ ）。

- A. 空格 B. Tab
C. \ D. 空格和 Tab

【解析】“\”符号为 Python 语言的换行符，不能用于缩进。答案为 C。

2. 以下关于 Python 语言中注释说法错误的是（ ）。

- A. 注释语句可以被执行
B. 注释语句以“#”符号开头
C. 多行注释可以用“""" 将其包围起来
D. 单行注释可以和非注释语句的同一行，并出现在非注释语句之后

【解析】Python 源代码运行时会跳过注释语句，而不是执行它。答案为 A。

3. 以下不能在 Python 交互式环境下得到运算结果的是（ ）。

- A. $2+6*(1.4+1)$ B. $6**(3+9)$
C. $6/\sin(2)$ D. $2(4-5*5)$

【解析】Python 中表达式和其他高级语言一样是不能省略乘法符号的。答案为 D。

4. 要输出如图 2-5 所示字符图形可以使用的代码（ ）。（^代表一个空格符号）

```

      *
    * *
  * * *

```

图 2-5 程序的执行结果

- A. `print('^'^'^*', '\n', '^*^*', '\n', '*^*^*')`
B. `print('^'^*', '\n', '^*^*', '\n', '*^*^*')`
C. `print('^'^'^*', '\n', '^*^*', '*^*^*')`
D. `print('^'^*', '\n', '^*^*', '\n', '***')`

【解析】print 函数会在输出多个值时自动在不同参数之间添加空格。答案为 A。



二、实验题

1. 在你选择的 Python 语言编辑器中输入本章中的缩进示例代码。
2. 请在 Python 的交互式环境下，首先定义 $x=2$ ，计算以下代数式的值并用 `print` 函数来输出：

$$x^2-4 \qquad 6(x/7) \qquad 6/(4+5x)$$

3. 请编程要求用户输入一个符号，并用所给的符号输出如图 2-6 所示字符示例图形。

【提示】用一个变量接收用户的输入符号，之后用 `print` 函数按要求输出。

```
*****  
*  
*****  
*  *  
*****
```

图 2.6 程序的输出结果

第 3 章 Python 数据类型

程序员开发程序的目的是进行数据处理，而日常生活中的数据各种各样，要想准确地处理数据，必须了解要处理数据对象的数据类型，才能在处理时根据其类型进行正确的运算。否则，不仅不能正常地处理数据，连程序也不能正常运行。

Python 语言的数据类型也很多，仅使用内置的数据类型就可以完成很多数据处理任务。对于简单的数据处理任务，根本无须自定义数据类型或数据结构。

本章主要讲解 Python 中的常用数据类型，内容包括：

- 字符串；
- 整数；
- 浮点数；
- 类型转换；
- 字符串编码；
- 列表、元组与字典；
- 序列及其通用操作；
- 相关逻辑运算。

3.1 Python 简单数据类型



Python 语言中的数据类型很多，主要有简单数据类型和结构数据类型。简单数据类型就是我们日常生活中经常使用的数据，本节主要介绍这些简单的数据类型。

3.1.1 字符串（str）

在第 1 章中的第一个 Python 程序 `hello.py` 中就已经使用过一次字符串。字符串主要用于存储和表示文本。Python 中的字符串通常由单引号 “`'`”、双引号 “`"`”、三个单引号或三个双引号包围的一串字符组成。



单引号和双引号都应当是英文字符中的符号。

字符串中的字符可以包含数字、字母、中文字符、特殊符号以及一些不可见的控制字符，如换行符、制表符等。例如以下都是字符串：

```
'abc'、'123'、"ab12*"、"大家"、'''123abc'''、"""abc123"""
```

字符串还可以通过序号（序号从 0 开始）来取出其中某个字符，例如 `'abcde'[1]` 取得的值为 `'b'`。

那么，既然都是字符串，这三种表示方法各有什么区别与联系呢？

单引号字符串与双引号字符串本质上是相同的。但当字符串内含有单引号时，如果用单引号字符串就会导致无法区分字符串内的单引号与字符串标志的单引号，就要使用转义字符串，如果用双引号字符串就可以在字符串中直接书写单引号即可，比如 `'abc"dd"ef'`、`"acc'd'12"`。

三引号字符串可以由多行组成，单引号或双引号字符串则不行，当需要使用大段多行的字



字符串就可以使用它。比如：

```
'''
This is a function.
Return a tuple.
'''
```

1. 转义字符串

在 Python 中如果要在字符串中包含控制字符或特殊含义的符号，就需要使用转义字符。常见的转义字符如下：

- `\n` 换行符；
- `\t` 制表符 (Tab)；
- `\r` 回车 (Enter)；
- `\\` “\” 字符；
- `\'` 单引号字符串中的单引号；
- `\"` 双引号字符串中的双引号。

比如以下字符串中都包含了转义字符：

```
'abc\nabc'          #包含一个换行符的字符串
'abc\'2\'abc'       #字符串中包含被单引号引起的2
```

如图 3.1 所示，在交互式环境中应用输出函数输出带有转义字符的字符串：

```
>>> print('aa\naa')
aa
aa
>>> print('aa\taa')
aa      aa
>>> print('aa\'2\'aa')
aa'2'aa
```

图 3.1 转义字符示例代码

2. 字符串运算

在 Python 中字符串是可以使用 “+” “*” 运算符进行运算的，如图 3.2 所示。

```
>>> 'aaa' + 'bbb'
'aaabbb'
>>> "Python " * 3
'Python Python Python '
```

图 3.2 字符串运算符演示

相信你一定看出来 “+” 就是连接字符串；“*” 就是单字符串的多次连接，这就是乘法的本来意义吧？

3. 字符串处理函数

除了用运算符对字符串进行运算外，Python 还提供了很多对字符串操作的函数，其中常用的字符串操作函数及其描述如表 3.1 所示。

表 3.1 常见的字符串函数及其描述

字符串操作	描 述
<code>string.capitalize()</code>	将字符串的第一个字母大写
<code>string.count()</code>	获得字符串中某一子字符串的数目

续表

字符串操作	描 述
<code>string.find()</code>	获得字符串中某一子字符串的起始位置, 无则返回-1
<code>string.isalnum()</code>	检测字符串是仅包含 0-9A-Za-z
<code>string.isalpha()</code>	检测字符串是仅包含 A-Za-z
<code>string.isdigit()</code>	检测字符串是仅包含数字
<code>string.islower()</code>	检测字符串是否均为小写字母
<code>string.isspace()</code>	检测字符串中所有字符是否均为空白字符
<code>string.istitle()</code>	检测字符串中的单词是否为首字母大写
<code>string.isupper()</code>	检测字符串是否均为大写字母
<code>string.join()</code>	连接字符串
<code>string.lower()</code>	将字符串全部转换为小写
<code>string.split()</code>	分割字符串
<code>string.swapcase()</code>	将字符串中大写字母转换为小写, 小写字母转换为大写
<code>string.title()</code>	将字符串中的单词首字母大写
<code>string.upper()</code>	将字符串中全部字母转换为大写
<code>len(string)</code>	获取字符串长度

其中 `split()` 函数返回以指定的字符将字符串分割成为列表 (稍后讲解) 形式并返回, 但并不改变原字符串, 它的原型如下:

```
split([sep [,maxsplit]])
```

其参数含义如下:

- `sep` 可选参数, 指定分割的字符, 默认为空格;
- `maxsplit` 可选参数, 分割次数。

`join()` 函数将原字符串插入参数字符串中的每两个字符之间。如果参数字符串中只有一个字符, 那么返回参数字符串。同样, `join()` 并不改变原字符串, 只是返回一个新的字符串。

【实例 3-1】 演示了部分字符串函数的应用, 代码如下:

```
mystr = 'Beautiful is better than ugly.'
print('source string is:', mystr)
print('swapcase demo\t', mystr.swapcase())
print('upper demo\t', mystr.upper())
print('lower demo\t', mystr.lower())
print('title demo\t', mystr.title())
print('istitle demo\t', mystr.istitle())
print('islower demo\t', mystr.islower())
print('capitalize demo\t', mystr.capitalize())
print('find demo\t', mystr.find('u'))
print('count demo\t', mystr.count('a'))
print('split demo\t', mystr.split(' '))
print('join demo\t', ' '.join('abcde'))
print('len demo\t', len(mystr))
```

【代码说明】 以上代码中从第三行开始, 每行都调用了一个字符串函数, 并打印出结果。

【运行效果】 该程序运行后的输出如图 3.3 所示。



```
>>>
source string is: Beautiful is better than ugly.
swapcase demo    bEAUTIFUL IS BETTER THAN UGLY.
upper demo       BEAUTIFUL IS BETTER THAN UGLY.
lower demo       beautiful is better than ugly.
title demo       Beautiful Is Better Than Ugly.
istitle demo     False
islower demo     False
capitalize demo   Beautiful is better than ugly.
find demo        3
count demo       2
split demo       ['Beautiful', 'is', 'better', 'than', 'ugly.']
join demo        a b c d e
len demo         30
```

图 3.3 字符串函数演示

3.1.2 整数 (int)



整数包括正整数、负整数和零，正如在第 2.3.2 小节中看到的，Python 中整数的范围是很大的。Python 中整数还能以几种不同的进制进行书写。0+“进制标志”+数字代表不同进制的数，进制标志有以下几种：

- 0o[0O]数字 表示八进制整数（例如：0O24、0O24）；
- 0x[0X]数字 表示十六进制整数（例如：0x3F、0X3F）；
- 0b[0B]数字 表示二进制整数（例如：0b101、0B101）。

不带进制标志的为十进制整数。



注意

每种进制开头是数字 0；八进制的数字 0 后是小写字母 o 或大写字母 O；但十进制不得以数字 0 开头书写；每种进制书写时数码不得超过进制规定的数码范围。

Python 语言中整数可参与的运算有很多，主要包括以下各种运算符（见表 3.2）。

表 3.2 整数运算符及其描述

运 算 符	描 述
**	乘方运算符
*	乘法运算符
/	除法运算符
//	整除运算符
%	取余运算符
+	加法运算符
-	减法运算符
	位或
^	位异或
&	位与
<<	左移运算
>>	右移运算

“//”运算符就是取商而丢弃余数，比如 $14 \div 4 = 3 \cdots 2$ ，所以 $14//4 = 3$ ；而“%”运算符的运算结果是余数而丢弃商，所以 $14\%4 = 2$ 。

|、^、&、<<、>>运算符都是位运算符，要依据其二进制形式进行运算，你可以参考相关

资料。

当同一个算式中含有表 3.2 中的多个运算符时, Python 会按照优先级进行运算: 即先计算优先级高的, 后计算优先级低的; 同级的运算符则从左向右计算。

表 3.2 中的运算符优先级从高到低排列如下:

- **
- *, /, %
- +, -
- |, ^, &, <<, >>

但是在同一个式子中, 你可以使用括号来修改运算符的优先级, 即括号内的具有高优先级。你不必强记运算符的优先级, 在没有把握的情况下可以运用括号。



“/”运算符的运算结果为浮点数, 即使是两个整数相除。

3.1.3 浮点数 (float)

浮点数就是常用的带小数的数, 当然整数部分也可以为零。浮点数的书写除了一般形式(形如 3.1415962) 外, 还有以下几种表示方法:

- 19. 小数部分为零, 可以不写;
- .098 整数部分为零, 可以不写;
- -2e3 科学计数法, 表示 -2×10^3 。

浮点数可以参加的运算支持表 3.2 中除位运算之外的运算符。

3.1.4 类型转换

在 Python 中常用的数据类型之间是可以相互转换的, 它们之间转换所使用的函数如下:

- str(object="") 可以将整数和浮点数转换为字符串, 默认建立空字符串;
- int(x, base=10) 将数字字符串或数值转换为整数 (base 表示数制);
- float(x) 将字符串或数值转换为浮点数。

此外, str() 可以创建一个空字符串。int() 也可以建立一个默认值为 0 的整数。float() 可以建立一个默认值为 0.0 的浮点数。

【实例 3-2】 演示常见类型转换函数的使用方法, 其代码如下:

```
print('int("23.5"):\t',int(23.9))           #浮点数->整数
print('int("23.001"):\t',int(23.001))       #浮点数->整数
print('int("23"):\t',int("23"))             #字符串->整数

print('float(3):\t',float(3))                 #整数->浮点数
print('float("3"):\t',float('3'))           #字符串->浮点数
print('float("3.2"):\t',float('3.2'))       #字符串->浮点数

print('str(23):\t',str(23))                   # 整数->字符串
print('str(23.3):\t',str(23.3))              # 浮点数->字符串

print('int("23.1"):\t',int("23.1"))         #错误的转换
```

【代码说明】 本实例中演示了各种常见的类型转换, 即整数、浮点数和字符串之间的相互转换, 其中最后一行会发生错误。





【运行效果】图 3.4 为本例程序运行的输出结果，前两行显示出浮点数转换为整数时始终将小数部分舍去，而不是四舍五入法。最后四行显示了程序在运行时发生了错误，提示信息大致意思是：字符串 23.1 不能被转换为整数。

```
>>>
int("23.5"):      23
int("23.001"):    23
int("23"):         23
float(3):          3.0
float("3"):        3.0
float("3.2"):      3.2
str(23):           23
str(23.3):         23.3
Traceback (most recent call last):
  File "D:/lx/a3_2.py", line 12, in <module>
    print('int("23.1"):\t',int("23.1"))
ValueError: invalid literal for int() with base 10: '23.1'
```

图 3.4 类型转换的运行结果



注意 将字符串形式的数值转换为整数时，其中只能包含数字。

本书的第 2.2.1 小节提到 input() 函数接收键盘输入时，无论输入的是整数、浮点数还是字符串，Python 得到的都是字符串，那么当需要整数或浮点数时都要进行类型转换。

在交互式环境下的示例代码如下：

```
>>> aint = input("Please input a int:") #调用输入函数接收键盘输入并用 aint 引用
Please input a int:34                  #用户通过键盘输入“34”
>>> aint                               #结果为字符串
'34'
>>> aint = int(aint)                  #将字符串转换为整数
>>> aint
34
>>> afloat = float(34)                #将字符串转换为浮点数
>>> afloat
34.0
```

3.2 字符串进阶

Python 语言的字符串还有一种特殊的形式和格式化方法，本节就介绍一下原始字符串表示方式及使用、如何格式化字符串和中文字符串的使用。

3.2.1 原始字符串

在本书的第 3.1.1 小节提到字符串中可以包含有一些转义字符，但它们都是以“\”开头，而当字符串中需要“\”字符时就必须写成“\\”形式，如果需要两个“\\”时，就必须写成“\\\\”的形式。Python 中有一种解决办法，可以不用写这么多个“\”字符，那就是使用原始字符串。原始字符串就是在字符串前以 r 或 R 作为标志，如图 3.5 所示，第一行语句为在交互式环境下演示的一个原始字符串，第二行可以看出字符串在显示时已经成为“\\”，即转义字符，而在原始字符串中只要使用一个“\”符号，这种表示法常用来在 Windows 系统下表示路径中的分隔符，如 r'c:\windows\system'，更常用的还有在本书后续章节中讲述的正则表达式。

```
>>> r'abc\akdd'
'abc\\akdd'
>>> r'abc\'
SyntaxError: EOL while scanning string literal
```

图 3.5 原始字符串示例



注意 原始字符串不得以“\”结尾。见图 3.5 所示的第三行（第二个示例）及后面错误提示。

3.2.2 格式化字符串

在 Python 中，所有字符串中的字符顺序是不可变的，但是在某些情况下，比如输出时，可能又要根据不同的需要修改字符串的内容，这时，可使用 Python 的格式化字符串功能。

在 Python 中，可以在字符串中使用以“%”开头的字符，以在程序中改变字符串中的内容，常用的格式化字符及其意义如表 3.3 所示。

表 3.3 格式化字符及其意义

格式化串	意 义
%c	单个字符
%d	十进制整数
%o	八进制整数
%s	字符串
%x	十六进制整数，其中的字母小写
%X	十六进制整数，其中的字母大写

如图 3.6 所示，在交互式环境下演示了在字符串中使用格式化字符的方法，让同一字符串可以根据变量的不同，显示不同的具体内容。

```
>>> print("I am a %s." % "programmer")
I am a programmer.
>>> print("%d + %d = %d" % (2,3,2+3))
2 + 3 = 5
>>> print("%d + %d = %d" % (4,6,4+6))
4 + 6 = 10
>>>
```

图 3.6 字符串格式化示例

3.2.3 中文字符串处理

在本书介绍的 Python 3.x 版本中，字符串已经全面支持中文了，并且默认的都是 utf-8 编码字符串。那么在不同的平台下或应用系统间，字符编码可能不同，这就会造成字符乱码的现象，这时应使用 Python 中的相关函数对字符串做相应的编码或解码。

比如，在网络上发送字符串时要求必须转换为字节串（bytes）形式，那么就要使用字符串的 encode() 方法，它返回的是字节串（bytes），其形式如下：

```
encode(encoding='utf-8', errors='strict')
```

参数意义如下：

- encoding 默认编码方式为 utf-8，也可以使用 gbk、gb2312 等方式编码；
- errors 编码错误的处理方式，默认为 strict（报错），也可以是 ignore、replace 等形式。

反之，如果从网络上接收的字节串（bytes）为字符串，则使用字节串（bytes）的 decode() 方法来进行解码，才能看到原来的字符串，其原型如下：

```
decode(encoding='utf-8', errors='strict')
```

其参数含义与前面 encode() 函数相同，返回的是字符串类型数据。



解码时的 encoding 要与编码时的 encoding 一致，否则将不能还原或报错。

如图 3.7 所示，在交互式环境下对字符串进行编码和解码的示例：

```
>>> wd = "我爱Python!"
>>> wd_utf8_bts = wd.encode()
>>> wd_utf8_bts
b'\xe6\x88\x91\xe7\x88\xb1Python!'
>>> wd_utf8_bts.decode()
'我爱Python!'
>>> wd_gb_bts = wd.encode('gb2312')
>>> wd_gb_bts
b'\xce\xd2\xb0\xaePython!'
>>> wd_gb_bts.decode('gb2312')
'我爱Python!'
>>> wd_gb_bts.decode('uft-8')
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    wd_gb_bts.decode('uft-8')
LookupError: unknown encoding: uft-8
```

图 3.7 字符串编码与解码示例

从图 3.7 中可以看出，`wd.encode()`就是调用字符串的编码函数，将其编码为 `bytes`，然后调用 `wd_utf8_bts.decode()`，即调用 `bytes` 的解码函数，还原了原来的字符串。比较 `wd_utf8_bts` 和 `wd_gb_bts` 可以看出，它们虽然都对同一字符串进行编码，但指定的编码形式不同，因此编码结果的 `bytes` 也不相同。所以编码和解码所指定的编码类型应该相同，这样才能正确还原字符串。

3.3 标志符与赋值号



标志符是任何一门高级编程语言所必须使用的，它主要用于变量名、类名和函数名等；“=”在一般的编程语言中都是赋值符号，其作用是给变量赋值。本节介绍编程语言中的这两项基本内容。

3.3.1 标志符

标志符是高级程序设计语言必须使用的用来代表数据的符号。Python 语言规定标志符只能以字母或下划线引导，其后跟随 0 个或多个非空格字符、下划线或数字，并且是大小写敏感的。它不能与 Python 语言的关键字（Python 语言中已经定义过的字）相同。

例如 `abc`、`a2c`、`_acd`、`aKb`，这些标志符都是合法的标志符，但是一般命名的标志符应该结合它在程序中的用途或意义进行命名，这样更有利于阅读和交流程序，比如 `age`、`name` 等。

标志符可以用来作为变量的标志符即变量名；也可以作为函数的名字即函数名；还可以作为类的名字。在程序设计中经常需要自定义变量名、函数名、类名等名称。



标志符是大小写敏感的，即若组成字母相同，只要大小写不同就是不同的变量名。

3.3.2 赋值号 “=”

如果你学习过其他程序设计语言，你一定知道这就是赋值符号。即将右边的值赋给左边的变量。你这样理解的话，对于编程的影响也不大。那么它的真正含义是什么呢？

在 Python 中, “=” 的作用是将对象引用与内存中某对象进行绑定。如果对象已经存在, 就简单地绑定, 以便引用右边的对象; 若对象不存在, 就由 “=” 操作符创建对象并绑定。

例如以下示例代码:

```
x = 3      #内存中还不存在 3, 则在内存中创建 3, 并将 x 与之绑定。此时 x 引用 3
y = 3      #内存已存在 3, 则直接将 y 与内存中的 3 进行绑定
           #此时 x 与 y 同时绑定了内存中同一个对象 3
y = x + 5   #此操作会计算右边的值为 8, 内存中不存在 8, 则在内存中创建 8, 将 y 绑定到它
```

Python 是一种动态类型机制的语言。所以变量在使用前不需要定义它的类型; 同时, 在任何时刻, 某个对象引用都可以重新引用一个不同 (类型) 的对象。

请看以下在交互式环境中的示例代码:

```
>>> x = 2
>>> y = 2
>>> x is y      #x 和 y 引用同一个对象, 所以结果为 True
True
>>> x = 'a'      #x 被重新绑定到一个字符串值上 (原来绑定的是整数 2)
>>> y = 3.45     #y 被重新绑定到一个浮点数值上 (原来绑定的是整数 2)
>>> x            #显示 x 绑定的值
'a'
>>> y            #显示 y 绑定的值
3.45
```

那么, 你可能怀疑这样会不会容易导致错误呢? 比如搞不清某个变量名引用的到底是什么类型的数据。Python 会在运行时执行类型检查的, 一旦不能运算, 会引发错误。

请看以下示例代码:

```
>>> x = 2
>>> y = 'a'
>>> x + y      #x 为整数, y 为字符串, 运算时引发错误。
Traceback (most recent call last): #引发错误
  File "<pyshell#39>", line 1, in <module>
    x + y
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

3.4 Python 结构数据类型

Python 语言中的结构数据类型有很多种, 但最常用的主要有列表、元组和字典。本节主要介绍这三种类型的结构数据。



3.4.1 列表 (list)

列表是最常见的一种数据形式, 它能把大量的数据放在一起, 对其进行集中处理。不仅可以方便地进行数据处理, 还可以减少声明很多变量。

列表是以方括号 “[]” 包围的数据集合, 不同成员间以 “,” (半角符号) 分隔。列表中可包含任何数据类型, 也可包含另一个列表。列表也可以通过序号来访问其中的成员。

在交互式环境下, 演示列表的创建和基本使用的示例代码如下:

```
>>> list()      #创建一个空列表
[]
>>> []          #创建一个空列表
[]
>>> [1,]        #创建一个只有一个元素的列表
[1]
>>> [1,2,3]     #创建一个有三个元素的列表
[1, 2, 3]
>>> alist = ['a',1,'b',2.0] #创建一个包含不同数据元素的列表
```



```
>>> alist[2]                #访问列表 alist 中的第 3 个元素（序号为 2）
'b'
>>> [1,2]+[3,4]             #列表支持加法运算
[1, 2, 3, 4]
>>> [None] * 5               #列表支持乘法运算
[None, None, None, None, None]
```



注意 列表元素的序号从 0 开始计数，即第一个元素的序号为 0。

Python 提供了对列表操作的多种操作函数，常用的操作函数及其描述如表 3.4 所示。

表 3.4 列表的操作函数及其描述

列表操作	描 述
list.append(x)	列表尾部追加成员 x
list.count(x)	返回列表中的参数 x 出现的次数
list.extend(L)	向列表中追加另一个列表 L
list.index(x)	返回参数 x 在列表中的序号（x 不存在则报错）
list.insert(index,object)	向列表中指定位置(index)插入数据(object)
list.pop()	删除列表中尾部的成员并返回删除的成员
list.remove(x)	删除列表中的指定成员（有多个则只删除第一个） 指定成员不存在则报错
list.reverse()	将列表中成员的顺序颠倒
list.sort()	将列表中成员排序（要求其成员可排序，否则报错）

以下是在交互式环境中对列表操作的示例代码：

```
>>> alst = [1,2,3,4,5]      #建立一个列表
>>> alst.append(1)          #列表尾部追加元素 1
>>> alst.count(1)           #统计 1 在列表中出现次数
2
>>> alst.extend([2,'insert']) #列表后追加另一个列表所有元素
>>> alst
[1, 2, 3, 4, 5, 1, 2, 'insert']
>>> alst.index(2)           #元素 2 在列表中首次出现序号
1
>>> alst.insert(3,0)        #在序号 3 处插入元素 0
>>> alst
[1, 2, 3, 0, 4, 5, 1, 2, 'insert']
>>> alst.pop()              #返回并删除列表最后一个元素
'insert'
>>> alst
[1, 2, 3, 0, 4, 5, 1, 2]
>>> alst.remove(1)          #删除列表中的元素 1（仅删除第一个）
>>> alst
[2, 3, 0, 4, 5, 1, 2]
>>> alst.reverse()         #列表内元素顺序颠倒
>>> alst
[2, 1, 5, 4, 0, 3, 2]
>>> alst.sort()             #对列表元素排序
>>> alst
[0, 1, 2, 2, 3, 4, 5]
```



注意 在列表操作中并不返回列表，而只是修改列表。

3.4.2 元组 (tuple)

元组可以看成是一种特殊的列表，与列表不同的是元组一旦建立就不能改变。既不能改变其中的数据项，也不能添加和删除数据项。因此，想让一组数据不能被改变就把它们放入到一个元组中即可，并且任何企图修改元组的操作都会发生错误。

元组的基本形式是以圆括号“()”括起来的数据元素，它也可以通过序号来引用其中的元素。

以下是在交互式环境下演示元组的使用示例代码：

```
>>> () #创建空元组
()
>>> tuple() #创建空元组
()
>>> (1,) #创建只有一个元素的元组
(1,)
>>> 2,3 #直接用逗号隔开两个值，可以创建一个元组
(2, 3)
>>> x,y=2,3 #右边为一元组，自动将元组第一个数值与 x 绑定，第二个与 y 绑定
>>> x
2
>>> y
3
>>> x,y=y,x #交换 x 与 y 的值（本质上右边是一个元组）
>>> x
3
>>> y
2
>>> atpl = (1,2,3)
>>> atpl[1] #引用元组序号为 1 的元素
2
>>> atpl[1]=3 #试图修改元组的元素，结果发生错误
Traceback (most recent call last):
  File "<pyshell#93>", line 1, in <module>
    atpl[1]=3
TypeError: 'tuple' object does not support item assignment
```



注意 建立只有一个元素的元组，元素后要有一个“,”。

3.4.3 字典 (dict)

字典是 Python 中比较特别的一类数据类型，字典中每个成员是以“键：值”对的形式存在的。

字典以大括号“{ }”包围的以“键：值”对方式声明和存在的数据集合。与列表的最大不同在于字典是无序的，其成员位置只是象征性的，在字典中通过键来访问成员，而不能通过其位置来访问该成员。

以下在交互式环境中演示 Python 中字典的创建与成员引用的示例代码：

```
>>> {} #建立空字典
{}
>>> dict() #建立空字典
{}
>>> adct = {'a':1,'b':2,'c':3.4}
>>> adct
{'a': 1, 'b': 2, 'c': 3.4}
>>> adct['a'] #用键名引用成员
1
```





```
>>> adct[1]                                     #试图用序号引用字典成员，结果发生错误
Traceback (most recent call last):
  File "<pyshell#99>", line 1, in <module>
    adct[1]
  KeyError: 1
```

Python 中也提供了很多有用的字典操作函数，如表 3.5 所示。

表 3.5 字典操作函数及其描述

字典操作	描 述
dic.clear()	清空字典
dic.copy()	复制字典
dic.get(k,[default])	获得键 k 对应的值，不存在则返回 default
dic.items()	获得由键和值组成的迭代器
dic.keys()	获得键的迭代器
dic.pop(k)	删除 k:v 成员对
dic.update(adict)	从另一个字典更新成员（不存在就创建，存在则覆盖）
dic.values()	获得值的迭代器
dic.fromkeys(iter,value)	以列表或元组中给定的键建立字典，默认值为 value
dic.popitem()	从字典中删除任一 k:v 项并返回它
dic.setdefault(k,default)	若字典中存在 key 值为 k 的，则返回其对应的值；否则，在字典中建立一个 k:default 字典成员

以下是在交互式环境下对字典操作的示例代码：

```
>>> adct = {'a':1,'b':2}                        #建立一个新字典
>>> adct
{'a': 1, 'b': 2}
>>> adct.get('a')                               #获取键“a”对应的值
1
>>> adct.get('d',0)                             #获取不存在的键“d”对应的值
0
>>> adct['d']                                    #直接以键获取值，不存在而发生错误
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
    adct['d']
  KeyError: 'd'
>>> adct.items()                                #返回字典所有的键值对
dict_items([('a', 1), ('b', 2)])
>>> adct.keys()                                  #返回字典的所有键
dict_keys(['a', 'b'])
>>> adct.values()                                #返回字典所有的值
dict_values([1, 2])
>>> adct.update({'b':4})                         #用另一个字典（存在键）去更新 adct 字典
>>> adct
{'a': 1, 'b': 4}
>>> adct.update({'c':2})                         #用另一个字典（不存在键）去更新 adct 字典
>>> adct
{'a': 1, 'b': 4, 'c': 2}
>>> adct.setdefault('a')
1
>>> adct.setdefault('d',0)
0
>>> adct
{'d': 0, 'a': 1, 'b': 4, 'c': 2}
>>> adct.pop('d')                                #删除 d 键值对并返回值
0
```

```
>>> adct
{'a': 1, 'b': 4, 'c': 2}
>>> adct.popitem()           #删除任一项键值对并返回它
('a', 1)
>>> adct
{'b': 4, 'c': 2}
>>> adct.pop('d')           #删除不存在的键值对, 发生错误
Traceback (most recent call last):
  File "<pyshell#51>", line 1, in <module>
    adct.pop('d')
KeyError: 'd'
```

3.5 内置常量与逻辑运算符、比较运算符



在 Python 语言中, 除了以上介绍的各种数据类型及相关的运算外, 还有一些常用的内置常量、逻辑运算、比较运算符, 下面来简单地介绍下。

3.5.1 常用内置常量

None 的意义正是其字面意义, 即“无”, 常用来表示没有值的对象。

True (真) 与 False (假) 是 Python 的逻辑型数据。

Python 中逻辑假包括 False、None、0、"" (空字符串)、() (空元组)、[] (空列表) 和 {} (空字典) 等, 而其余任何值都视为真。

3.5.2 Python 中逻辑运算符

Python 中逻辑运算符包括与 (and)、或 (or)、非 (not)。

not 的运算对象只有一个, 一般也称为一元运算符, 其规则是非假即真; 非真即假。例如:

- not False 值为 True;
- not () 值为 True;
- not 3 值为 False。

or 即“或”运算符, 两个参与运算的操作数有一个为真, 则结果为真, 否则结果为假。它是一种短路运算符, 并且总是返回决定运算结果的参与运算的操作数。其运算处理过程是这样的: 如果第一个操作数或表达式为真则直接返回第一个操作数, 而不处理第二个操作数或表达式; 如果第一个操作数或表达式为假则返回第二个操作数或表达式的值。

在交互式环境下使用 or 运算符的示例代码如下:

```
>>> [1,2] or 0        #第一个操作数为[1,2], 其逻辑值为真, 所以直接返回[1,2], 结果为真
[1, 2]
>>> 0 or (1,2)        #第一个操作数 0 代表假, 所以直接返回第二个操作数, 即 (1,2), 结果为真
(1, 2)
>>> [] or ()          #第一个操作数[] (空列表) 代表假, 直接返回第二个操作数() (空元组)
()
                      #其结果为假
```

and 即“与”运算符, 两个参与运算的操作数都是真, 则结果为真, 否则结果为假。它与 or 运算符相似, 也是一种短路运算符, 并且总是返回决定运算结果的参与运算的操作数。其运算处理过程是这样的: 如果第一个操作数或表达式为假则直接返回第一个操作数, 而不处理第二个操作数或表达式; 如果第一个操作数或表达式为真则返回第二个操作数或表达式的值。

在交互式环境下的 and 运算符使用的示例代码如下:

```
>>> [1,2] and 3        #第一个操作数为[1,2]代表真, 返回第二个操作数 3, 结果为真
3
>>> [] and [1,2]        #第一个操作数为[], 代表假, 直接返回[], 结果为假
```



```
[ ]
>>> [ ] and ( )      #第一个操作数为[ ], 代表假, 直接返回[ ], 结果为假
[ ]
>>> 1 and 0           #第一个操作数为1, 代表真, 直接返回第二个操作数0, 结果为假
0
>>> 0 and True        #第一个操作数为0, 代表假, 直接返回0, 结果为假
0
```

3.5.3 Python 中比较运算符

Python 中比较运算符及其意义如表 3.6 所示。

表 3.6 比较运算符及其意义

运 算 符	意 义
==	相等
>	大于
<	小于
>=	大于或等于
<=	小于或等于
!=	不等于

这些运算符的意义和数学上相同, 有趣的是, Python 语言中允许连接使用这些比较运算符, 表示两个比较运算都成立时结果才为真。

以下为在交互式环境下的运算示例代码:

```
>>> 1<2<=3
True
>>> 'c'<'a'=='a'
False
>>> 'c'<'d'!=3
True
```



注意 字符(串)也是可以进行大小比较的, 其比较依据的是其 ascii 码。

3.5.4 Python 中其他逻辑操作符

Python 中除了基本的与、或、非逻辑运算符以外, 还有两类运算结果为逻辑型值的运算符。

1. is 和 is not

有文献资料中称它们为身份操作符, is 和 is not 都是二元操作符, 用于判断左端与右端对象引用是否指向同一个对象。对于 is 操作符, 相同则返回 True, 不同的则回 False, is not 操作符则相反。

在交互式环境下这两个操作符的使用示例代码如下:

```
>>> x = 3.14
>>> y = x
>>> x is y           #x 与 y 引用同一个对象, 结果为 True
True
>>> x is not y
False
>>> x is None         #x 为 3.14, 不指向 None, 结果为 False
False
>>> x = None
```

```
>>> x is None      x 为 None, 结果为真
True
```

2. in 和 not in

`in` 和 `not in` 称为成员操作符,用于检查某个数据是否存在于某包含多个成员的数据类型(如字符串、列表、元组、字典等)之中。如果是成员关系,则 `in` 返回真;否则返回假;而 `not in` 则相反。

在交互式环境下这两个操作符的使用示例代码如下:

```
>>> alst = [1,2,3]
>>> atpl = ['a','b','c']
>>> 1 in alst                #整数 1 是 alst 的成员, 返回 True
True
>>> 1 in atpl                #整数 1 不是 atpl 的成员, 返回 False
False
>>> 'a' not in atpl          #'a' 是 atpl 的成员, 返回 False
False
>>> 'a' not in alst          #'a' 不是 atpl 的成员, 返回 True
True
>>> 'a' in 'abcde'           #'a' 是 'abcde' 的成员, 返回 True
True
>>> adct = {'a':1,'b':2}
>>> 'a' in adct              #检查 'a' 是否在字典的键成员, 返回 True
True
>>> 1 in adct                #1 是字典中的值, 而不是字典的键成员, 返回 False
False
```



注意 成员操作符对于字典来说检查的是键成员而不是值成员。

3.6 序列

序列表示索引为非负整数的有序对象集合,包括前面所介绍的字符串、列表和元组。字符串是字符的序列,列表和元组则是任意 Python 数据类型或对象的序列。元组是不可变的,字符串也是不可变的(修改字符串就是重新创建一个字符串)。

3.6.1 序列切片

对于任何一个序列,它们的元素都是有序的,都可以使用序号来获取每一项成员的值,这在前面都已讲述过。另一方面 Python 中序列的序号既可以从左至右地从 0 开始计数,又可以由右至左地从 -1 开始计数。如图 3.8 所示,序列共有 8 个元素,从左至右数为 0 至 7,从右至左数为 -1 至 -8。

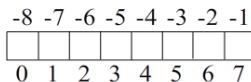


图 3.8 序列序号图

因此,以序号来取序列成员的话,同一成员可以有两个序号取法。

以下在交互式环境下使用序号取其成员的示例代码如下:

```
>>> alst = [0,1,2,3,4,5,6,7]
>>> alst[2]
2
>>> alst[-6]
2
```




序列的切片是指用形如[start:end:step]来取序列中的部分成员数据项。意思是从序列中 start 序号开始到 end 前一个结束，每隔 step 个取一个成员。当然并不是每次切片都要用 start、end、step。具体常用方法及其意义如表 3.7 所示（设 alst = [0,1,2,3,4,5,6,7]）。

表 3.7 切片常用的方法及其意义

使用形式	意 义
alst[:]	取全部成员数据项
alst[0:]	取全部成员数据项
alst[:-1]	取除最后一个成员之外的所有成员数据项
alst[2:5]	得到[2,3,4]
alst[::2]	每隔 1 个取一个成员，得到[0,2,4,6]
alst[0:5:2]	从 0 至 4 每隔一个取一项，得到[0,2,4]
alst[::-1]	从右至左取全部成员，得到[7, 6, 5, 4, 3, 2, 1, 0]
alst[5:0:-2]	从 5 至 0（不包括 0）从右至左每隔一个成员取一个成员，得到[5, 3, 1]

在交互式环境下使用切片的示例代码如下：

```
>>> alst = [0,1,2,3,4,5,6,7]
>>> alst[:]
[0, 1, 2, 3, 4, 5, 6, 7]
>>> alst[0:]
[0, 1, 2, 3, 4, 5, 6, 7]
>>> alst[:-1]
[0, 1, 2, 3, 4, 5, 6]
>>> alst[2:5]
[2, 3, 4]
>>> alst[::2]
[0, 2, 4, 6]
>>> alst[0:5:2]
[0, 2, 4]
>>> alst[::-1]
[7, 6, 5, 4, 3, 2, 1, 0]
>>> alst[5:0:-2]
[5, 3, 1]
```

切片的方法是非常灵活的，多实践，多体验，就能熟练使用了。



注意 切片所取的元素从 start 开始，到 end 前一个结束，不包括序号为 end 的元素。

3.6.2 序列内置操作

在 Python 中已经内置了一些对序列进行操作的方法，如表 3.8 所示。

表 3.8 序列操作方法

方 法	操作方法
len(s)	返回 s 的元素数（长度）
min(s)	返回 s 中的最小值
max(s)	返回 s 中的最大值
sum(s[,start])	返回 s 中各项的和
all(s)	s 中所有项为真，则返回真，否则返回假
any(s)	s 中有一项为真，则返回真，否则返回假

以下为在交互式环境下运用的示例代码：

```
>>> alst = [0,1,2,3,4]
>>> len(alst)          #alst 共有 5 个元素
5
>>> min(alst)
0
>>> max(alst)
4
>>> sum(alst)
10
>>> sum(alst,15)        #求所有元素的和，连同 15 一起加起来
25
>>> all(alst)
False
>>> any(alst)
True
```



注意

min()、max()函数要求序列中的元素能比较大小；sum()函数要求序列中元素只能是 int 或 float 类型。

3.7 小结

本章主要讲述了 Python 的简单数据类型（整数、浮点数、字符串）、结构数据类型（列表、元组和字典）、它们之间的类型转换、它们可参与的运算和相关函数，另外还包括标志符、常用内置常量、逻辑运算等内容。

通过本章学习，应掌握整数、浮点数、字符串及其运算和相关函数，列表、元组、字典及其运算和相关函数，整数、浮点数、字符串的类型转换，标志符与值的绑定，原始字符串，字符串的格式化与编码，常用内置常量，逻辑运算符与比较运算符、序列切片、序列内置操作等内容。

3.8 本章习题

一、选择题

1. 以下在 Python 中为正确字符串的是（ ）。

- A. 'abc"dd" B. 'abc"dd'
- C. "abc"dd" D. "abc\"dd\"

【解析】字符串两端需要相同的符号，如内部有与两端符号相同的则要转义。答案为 B。

2. 'ab'+c*2 的结果是（ ）。

- A. 'abc2' B. 'abcabc'
- C. 'abcc' D. 'ababcc'

【解析】先算乘法，后算加法。答案为 C。

3. 以下运算结果为假的是（ ）。

- A. '36'.isalnum() B. '36'.isdigit()
- C. '36'.islower() D. 'ab'.isalnum()

【解析】‘36’不是字母，结果为假。答案为 C。

4. 以下不是 Python 中整数的是（ ）。

- A. 0897 B. 0x57



C. 987 D. 0b1011

【解析】八进制最大数为7，所以A不是，答案为A。

5. 以下结果不是Python中数值的是（ ）。

A. 0897 B. -2e9
C. 987 D. float('3.a4')

【解析】float()的参数只能包含数字和小数点，或者为科学计数法。答案为D。

6. 以下不能得到Python中整数的是（ ）。

A. 0897 B. 0x57
C. 987 D. 0b1011

【解析】Python中十进制整数不能以0开头。答案为A。

7. 以下Python语句会出错的是（ ）。

A. '你好'.encode()
B. '你好'.decode()
C. '你好'.encode().decode()
D. 以上都不会出错

【解析】字符串不能再解码。答案为B。

8. 以下Python语句会出错的是（ ）。

A. [2,3,4][2] = 5 B. (2,3,4)[2] = 5
C. {'a':3,}['a'] = 8 D. {'a':3,}.get('b')

【解析】元组是不可变的，所以不能修改其所包含的值。答案为B。

9. 以下Python表达式返回为True的是（ ）。

A. 3 and 1 or 4 B. not 0
C. 3<4>5 D. 1 not in [1,2,3]

【解析】A项返回的为1，而not总是返回True或False。答案为B。

10. [0,1,2,3][1:3]返回的是（ ）。

A. [0,1,2,3] B. [1,2,3]
C. [1,2] D. [0,1,2]

【解析】切片包括“:”前序号对应的元素，而不包含“:”后序号对应的元素。答案为C。

二、实验题

1. 编程实现用户输入1个整数，输出这个整数的平方。

2. 编程实现用户输入3个整数，放入列表，并输出其最小值。

【提示】对于用户的输入，应先转换为整数，再放入列表，使用内建函数min来取得最小值。

3. 编程实现用户输入5个整数，并得到列表[0,1,2,3,4]，然后选用两种切片方法取出列表中的[1,3]并输出。

4. 用三种方法编程实现初始化字典{'a':1,'b':2}，并设置其key为'c'的对应值为3。

【提示】可以使用直接建立字典、用dict函数来初始化一个空字典后加入元素、用dict函数转换两个列表为字典。

5. 输入某科目排名前10的分数，求其总分和平均分。

【提示】对于用户的输入，应先转换为整数，再放入列表；使用内建函数sum来计算总分，用内建函数len计算总人数，之后求出平均分。

第4章 控制语句执行流程

当你观察日常生活中的所需要完成的工作时，你会发现：有的工作就是一个动作序列从头至尾执行完就行了；有的工作则是根据不同的情况进行不同的处理；有的工作则是简单机械地重复某个或某些工作序列。

而计算机程序在解决某个具体的问题时，也包括以上三种情形。即顺序执行所有的语句、选择执行部分语句和循环执行部分语句，正好对应着程序设计中的三种程序执行结构流程：顺序结构、选择结构和循环结构。

事实证明，任何一个能用计算机解决的问题，只要应用这三种基本结构来写出的程序都能解决，Python 语言当然也具有这三种基本结构。

本章主要介绍 Python 中的流程控制语句，内容包括：

- 选择结构 (if)；
- 用 for 循环；
- for 与 range()；
- 用 while 循环；
- 增量赋值运算符；
- 推导或内涵。

4.1 用 if 选择执行语句



前文所介绍的程序或示例，全部都是按照语句书写的顺序依次执行，当每条语句执行完一次之后，程序就自然结束了。而为了解决实际问题的程序，还需要依据程序执行过程中的某个条件来选择是否执行某一部分语句，抑或是选择执行两（多）部分语句的其中一部分，这就需要用到 if 语句。

4.1.1 if 基础

if 语句的作用是选择执行语句，其最简单的形式如下：

```
if <条件>:  
    <语句>
```

其基本语义是当条件为真时，执行其后缩进的语句；当条件为假时，跳过其后缩进的语句。其中的条件可以是任意类型的表达式。



注意 条件之后必须有“:”。

【实例 4-1】演示了一个最简单的 if 结构的语句，代码如下：

```
x = input('Please input a integer:')    #输入一个整数  
x = int(x)                             #字符串转换为整数  
if x < 0:                               #x<0 条件成立时执行其后缩进的语句  
    x = -x  
print(x)                                #输出 x 的值
```

【代码说明】这是一个用于输出用户输入的整数绝对值的程序。其中 `x=-x` 是被选择执行的



语句。

【运行效果】代码运行效果如图 4.1 所示，用户输入-4，则输出其绝对值 4。

```
>>>
Please input a integer:-4
4
```

图 4.1 基本 if 语句的运行结果

然而，有时需要的是两部分语句，根据一个条件来选择其一执行，那么使用以下形式的 if 语句：

```
if <条件>:
    <语句 1>
else:
    <语句 2>
```

显然，它的基本语义是：当条件成立时执行语句 1，否则执行语句 2。

【实例 4-2】演示了一个有两条语句选择执行的 if 结构语句，代码如下：

```
x = input('Please input a integer:')    #输入一个整数
x = int(x)                             #字符串转换为整数
if x < 0:                               #x<0 条件成立时执行其后缩进的语句
    print('你输入了一个负数。')
else:                                   #当 x<0 条件不成立时执行其后缩进的语句
    print('你输入了一个零或正数。')
```

【代码说明】这是一个用于输出用户输入的整数符号的程序。其中的两个缩进的 print() 函数是被选择执行的语句。

【运行效果】代码运行效果如图 4.2 所示，两次程序运行分别输入了-1、+4，其输出则根据条件 ($x < 0$) 的成立与否分别输出了“你输入了一个负数。”和“你输入了一个零或正数。”。

```
>>>
Please input a integer:-1
你输入了一个负数。
=====
>>>
Please input a integer:+4
你输入了一个零或正数。
```

图 4.2 判断用户输入整数的两次运行结果

在刚才的实例 4-2 中，如果要分清正整数、负整数和零，那么可以使用以下形式的 if 语句结构：

```
if <条件 1>:
    <语句 1>
elif <条件 2>:
    <语句 2>
else:
    <语句 3>
```

此时，条件 1 成立时执行语句 1；条件 2 成立时执行语句 2；条件 1 和条件 2 都不成立时则执行语句 3。

【实例 4-3】演示了一个有两个条件判断和三条语句选择执行的 if 结构语句，代码如下：

```
x = input('Please input a integer:')    #输入一个整数
```

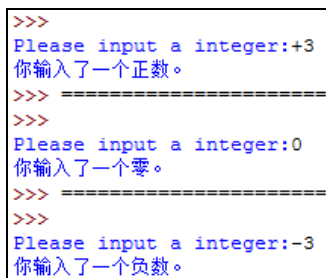
```

x = int(x)                                #字符串转换为整数
if x < 0:                                  #x<0 条件成立时执行其后缩进的语句
    print('你输入了一个负数。')
elif x == 0:                              #当 x 等于 0 时, 执行其后缩进语句
    print('你输入了一个零。')
else:                                     #当以上条件都不成立时执行其后缩进的语句
    print('你输入了一个正数。')

```

【代码说明】这是一个用于输出用户输入的整数是正数、负数、零三种情形的程序。其中的三个缩进的 print() 函数是被选择执行的语句。

【运行效果】代码运行效果如图 4.3 所示, 两次程序运行分别输入了 +3、0、-3, 其输出则根据条件 1 ($x < 0$)、条件 2 ($x == 0$) 的成立与否分别输出了“你输入了一个正数。”和“你输入了一个零。”以及“你输入了一个负数。”。



```

>>>
Please input a integer:+3
你输入了一个正数。
=====
>>>
Please input a integer:0
你输入了一个零。
=====
>>>
Please input a integer:-3
你输入了一个负数。

```

图 4.3 三部分语句选择执行

其实, 你还可以在 if 语句中加入更多的 elif 的条件选择分支, 根据需要在多个语句之间来选择其一执行, 形式如下:

```

if <条件 1>:
    <语句 1>
elif <条件 2>:
    <语句 2>
elif <条件 3>:
    <语句 3>
.....
else:
    <语句 n>

```

这种多分支的条件语句, 当条件 1 成立时执行语句 1; 条件 2 成立时执行语句 2; 条件 3 成立时执行语句 3……条件都不成立时执行语句 n。

【实例 4-4】演示了一个具有多个条件分支的 if 语句的使用, 代码如下:

```

x = input('输入你的总分:')                # 输入一个总分
x = float(x)                               # 字符串转换为浮点数
if x >= 90:                                # 当 x >= 90, 执行其后缩进的语句
    print('你的成绩为: 优。')
elif x >= 80:                              # 当 x >= 80, 执行其后缩进语句
    print('你的成绩为: 良。')
elif x >= 70:                              # 当 x >= 70, 执行其后缩进语句
    print('你的成绩为: 中。')
elif x >= 60:                              # 当 x >= 60, 执行其后缩进语句
    print('你的成绩为: 合格。')
else:                                     # 当以上条件都不成立时执行其后缩进的语句
    print('你的成绩为: 不合格。')

```



【代码说明】本实例代码中具有多个 `elif` 语句分支，根据每个条件的成立与否来选择输出你的成绩等次。

【运行效果】如图 4.4 所示，通过两次程序的运行分别输入的部分是 84 和 63，输出的结果如图 4.4 所示。

```
>>>
输入你的总分:84
你的成绩为:良。

>>> =====
>>>
输入你的总分:63
你的成绩为:合格。
```

图 4.4 多 `elif` 语句的运行结果



注意 当使用这种具有多个 `elif` 语句的分支结构时，应把握好多个条件语句之间的关系。只要有一个条件成立，就会将其后的一个部分语句执行后退出整个 `if` 语句。

如果实例 4-4 将 `elif` 分支条件 `x>=60` 和 `x>=70` 互换，则如下：

```
x = input('输入你的总分:')          #输入一个总分
x = float(x)                          #字符串转换为浮点数
if x >= 90:                            # 当 x >= 90，执行其后缩进的语句
    print('你的成绩为: 优。')
elif x >= 80:                          # 当 x >= 80，执行其后缩进语句
    print('你的成绩为: 良。')
elif x >= 60:                          #当 x >= 60，执行其后缩进语句
    print('你的成绩为: 合格。')
elif x >= 70:                          #当 x >= 70，执行其后缩进语句
    print('你的成绩为: 中。')
else:                                  #当以上条件都不成立时执行其后缩进的语句
    print('你的成绩为: 不合格。')
```

则这个程序中的 `print('你的成绩为: 中。')` 语句将永远不会被运行。因为当你输入 `[70, 80)` 这个区间中的数时先判断 `x>=60` 也是成立的，此时选择执行其后的语句并输出“你的成绩为合格。”，之后退出 `if` 语句，显然这个程序的运行结果没有完全达到你的期望。

此外，`if` 语句可以使用在一个单语句之中，实现三元运算符，基本形式如下：

<表达式 1> `if` <条件> `else` <表达式 2>

其语义是当条件为真时取得表达式 1 的值，否则取得表达式 2 的值。如以下代码中应用了这种形式的语句：

```
a = None
b = 3
x = b if a is not None else 0          #当 a 的值不为 None 时，x 取值为 3，否则 x 取值为 0
```

这样，本来需要多行的 `if` 语句完成的赋值，只需要一行就解决了，程序既简单又清晰。

4.1.2 `if` 语句的嵌套

在程序设计中，各种结构的语句嵌套的出现是难免的，当然 `if` 语句自身也存在着嵌套情况。嵌套的 `if` 语句和不嵌套的 `if` 语句在写法上的区别就是缩进的不同而已，以下为一种嵌套的 `if` 语句的写法：

```

if <条件>:
    if <条件>:                #此句开始的以后共 4 行都属于第一个 if 条件的嵌套，注意及缩进量大小
        <语句 1>
    elif <条件>:
        <语句 2>
else:
    <语句 3>

```

当然，你还可以可写很出多种形式不同的嵌套和多重嵌套。

【实例 4-5】演示了一个两层嵌套的 if 语句，代码如下：

```

a = int(input('请输入一个整数: '))
if a>0:
    if a>10000:                #此行开始及其后三行为一个 if 语句
        print("无法表示")
    else:
        print("可以表示")
    print("且大于 0")          #此行缩进量比上一行减少，因此不属于嵌套内的 if
else:
    if a<-10000:
        print("无法表示")
    else:
        print("可以表示")
    print("且小于 0")          #如果此行增加缩进量，则属于嵌套内的 if 语句

```

【代码说明】代码中首先根据其大于 0 还是小于 0 分为两个程序分支，然后在大于 0 分支中以大于 10000 为条件分成两个分支；小于 0 分支中以小于 -10000 为条件分成两个分支。

【运行效果】如图 4.5 所示，该程序共运行了 4 次，每次输入的数据决定程序运行了不同的分支。

```

>>>
请输入一个整数: 10234
无法表示
且大于0
>>> =====
>>>
请输入一个整数: 56
可以表示
且大于0
>>> =====
>>>
请输入一个整数: -234
可以表示
且小于0
>>> =====
>>>
请输入一个整数: -10234
无法表示
且小于0

```

图 4.5 嵌套 if 语句运行

在程序设计中，语句的嵌套一般不要太深，可以适当地修改多层嵌套的语句以减少嵌套层次，从而方便阅读和理解程序，但有时为了逻辑清晰也不用有意为之。如实例 4-5 可以改为不嵌套的 if 语句来完成相同的功能，代码如下：

```

a = int(input('请输入一个整数: '))
if a>10000:
    print("无法表示")
elif 0<a<=10000:
    print("可以表示的正数")
elif a<-10000:

```




```
print("无法表示")
else:
    print("可以表示的负数")
```

4.2 用 for 循环执行语句



for 语句是 Python 语言中构造循环结构程序的语句之一，在 Python 语言中，绝大多数的循环结构都是用 for 语句来完成的。本节主要介绍的就是 for 语句。

4.2.1 for 基础

Python 语言中的 for 语句与其他高级程序设计语言有很大的不同，其他高级语言 for 语句要用循环控制变量来控制循环。Python 中 for 语句是通过循环遍历某一序列对象（如第 3 章所述的元组、列表、字典等）来构建循环，循环结束的条件就是对象被遍历完成。

for 语句的形式如下：

```
for <循环变量> in <遍历对象>:
    <语句 1>
else:
    <语句 2>
```

for 语句的语义是遍历 for 语句中的遍历对象，每次循环，循环变量会得到遍历对象中的一个值，你可以在循环体中处理它；一般情况下，当遍历对象中的值全部用完时，就会自动退出循环。语句 1 就是 for 语句中的循环体，它的执行次数就是遍历对象中值的数量，else 语句中的语句 2 只在循环正常退出（遍历完所有遍历对象中的值）时执行。

【实例 4-6】演示了一个基本的 for 循环构建的语句，代码如下：

```
for i in [1,2,3,4,5]:          #i 是循环变量，每次循环，i 会依次取得后面列表中一个值
    print(i,"的平方是: ",i*i)  #此处理缩进的语句就是循环体
else:                          #for 循环正常结束时，其后缩进的语句就会执行
    print('循环结束!')
```

【代码说明】本例中的 for 语句遍历了一个包含有五个元素（1、2、3、4、5）的列表，所以循环变量 i 会依次取得 1、2、3、4、5 并输出其平方，当循环正常结束时，else 后缩进的语句就会执行，并输出“循环结束!”。

【运行效果】运行结果如图 4.6 所示，分别输出 1 至 5 的平方，最后输出“循环结束”。

```
>>>
1 的平方是: 1
2 的平方是: 4
3 的平方是: 9
4 的平方是: 16
5 的平方是: 25
循环结束!
```

图 4.6 for 语句的运行结果

4.2.2 for 语句与 break 语句、continue 语句

break 语句的作用是中断循环的执行，如果在 for 循环中执行了 break 语句，for 语句的遍历就会立即终止，即使还有未遍历完的数据，还是会立即终止 for 循环语句。

continue 语句的作用是提前停止循环体的执行，开始下一轮循环。在 for 语句中如果执行了 continue 语句，则 continue 语句后的循环体语句不会被执行，即提前结束了本次循环，然后

进入下一个遍历循环。

【实例 4-7】演示了 `break` 和 `continue` 语句的作用，代码如下：

```
for i in [1,2,3,4,5]:
    print(i)
    if i == 2:
        continue
    print(i,"的平方是: ",i*i)
    if i == 4:
        break
    else:
        print('循环结束!')
```

#当 i 等于 2 时，执行其后缩进的 `continue` 语句
#执行本语句后，其后的 `if` 语句和 `print()` 不会执行

#当 i 等于 4 时，执行其后缩进的 `break` 语句
#执行 `break` 时，终止循环
#因为 `for` 循环中的 `break` 语句执行中断了 `for` 循环
#该语句不会被执行

【代码说明】在该实例的代码中，当 `i` 遍历至 2 时，执行 `continue` 语句，使 2 的平方不会被输出；当 `i` 遍历至 4 时，执行 `break` 程序而中断了 `for` 循环，这样 `for` 循环就不是正常结束的，在 `else` 中的语句也就不会被执行了。

【运行效果】如图 4.7 所示，因为 `continue` 语句的执行，2 的平方没有被输出；因 `break` 语句的执行，使得 `for` 语句循环中止而没有遍历到列表中的 5，也就没有了 `else` 分支语句的执行及输出。

```
>>>
1
1 的平方是: 1
2
3
3 的平方是: 9
4
4 的平方是: 16
```

图 4.7 `break`、`continue` 示例程序输出

`break` 语句主要应用于当检测到某个条件时及时退出循环，从而提前结束循环的执行，以节约程序的运行时间。从某种程度上说，`continue` 语句也有着同样的作用。例如，要判断某个整数 `n` 是否为质数（素数），就要分别用 2 至 `n-1` 来除 `n`，只要有一个余数为 0 则可以判定其不是质数。使用 `break` 语句可以在检查到第一个能整除的数时结束循环，不用继续检查其后的所有值。

`for` 语句遍历列表、元组、字符串的基本形式是相同的，但对字典的遍历有些不同。因为字典既有键又有值，在遍历时不能直接对字典进行遍历而是通过字典的 `items()`、`keys()`、`values()` 等方法分别遍历其键和值、键、值。如果同时遍历键和值，在遍历时可以使用两个循环变量来分别接收键和值。

【实例 4-8】演示了用 `for` 遍历字典的实例，代码如下：

```
adct = {'apple':15,'banana':20,'pear':35}
for key,value in adct.items():
    print(key,':',value)

for key in adct.keys():
    print(key)

for value in adct.values():
    print(value)
```

#同时遍历键和值

#只遍历键

#只遍历值

【代码说明】代码中使用了三个 `for` 语句：第一个应用字典的 `items()` 方法依次同时遍历了字典的键和值，并且同时使用两个循环变量；第二个应用字典的 `keys()` 方法依次遍历了字典的



键：第三个应用了字典的 `values()` 方法遍历了字典的值。

【运行效果】如图 4.8 所示，前三行的输出结果是同时遍历键和值，其后的输出是分别遍历字典的键和值。

```
>>>
apple : 15
pear : 35
banana : 20
apple
pear
banana
15
35
20
```

图 4.8 for 遍历字典的运行结果

4.2.3 for 语句与 range() 函数

for 语句中的对象集合可以是列表、字典以及元组等，也可以通过 `range()` 函数产生一个整数列表，以完成计数循环。`range()` 函数的原型如下：

```
range([start,] stop[, step])
```

其参数含义如下：

- `start` 可选参数，起始数，默认值为 0；
- `stop` 终止数，如果 `range` 只有一个参数 `x`，那么 `range` 生产一个从 0 至 `x-1` 的整数列表；
- `step` 可选参数，步长，即每次循环序列增长值。



注意 产生的整数序列的最大值为 `stop-1`。

【实例 4-9】演示了用 for 语句和 `range()` 函数构建的两个循环结构的语句，代码如下：

```
print('第一次循环输出：')
for i in range(4):
    print(i)
print('第二次循环输出：')
for i in range(0,7,2):
    print(i)
```

【代码说明】代码中第一个循环语句中调用了 `range()` 函数，只给了一个参数 4，即会遍历从 0 至 3，共 4 个数；第二个循环调用了 `range(0,7,2)` 函数，会遍历 0、2、4、6 共 4 个数。

【运行效果】如图 4.9 所示，第一个循环输出了 0、1、2、3；第二个循环输出了 0、2、4、6。

```
>>>
第一次循环输出：
0
1
2
3
第二次循环输出：
0
2
4
6
```

图 4.9 用 `range()` 函数产生迭代



for 语句使用 range() 函数可以构建基于已知循环次数的循环程序，也可以以 range() 生成的数字作为索引来访问列表、元组、字符串中的值，还可以对遍历的序列实行处理，以得到相关的数据。



注意 range() 函数并不是在调用时一次生成整个序列，而是遍历一次才产生一个值，以减少内存的占用，其本质是一个迭代器（本书稍后会介绍）。

同 if 语句的嵌套一样，for 语句也是允许嵌套使用的。

【实例 4-10】演示了一个计算并输出由用户指定的两个整数之间所有素数的程序，代码虽然还不超过十行，但也比较复杂。其代码如下：

```
x = (int(input("请输入开始值（整数）：")),int(input("请输入一个结束值（整数）：")))
                                #输入两个整数，并放入一个元组中
x1 = min(x)                    #获取两个整数中较小的一个整数
x2 = max(x)                    #获取两个整数中较大的一个整数
for n in range(x1,x2+1):       #用 range() 产生从 x1 至 x2 的序列
    for i in range(2,n-1):     #用 range() 产生从 2 至当前值 n-1 的序列
        if n % i == 0:        #如果余数为 0 则执行其后缩进的 break 语句，中断内循环
            break
    else:                       #循环中断，则表明至少有一个整数因子，不是素数，不输出
        print(n,"是素数。")
```

【代码说明】代码中首先使用输入函数获取用户指定的序列开始和结束，然后用 for 语句构建了两层嵌套的循环语句用来求素数并输出。外循环用来产生待判断是否为素数的序列，内循环用来产生测试的因子。



注意 代码中的 else 子句的缩进表示它属于内嵌的 for 语句。如果多缩进一个单位，则属于其中的 if 语句；如果少缩进一个单位，则属于外层 for 语句。因此，Python 中的缩进是程序很重要的构成部分。

【运行效果】图 4.10 展示了程序的一次运行，用户输入了 275 和 300，程序运行完成后，输出了 275~300 之间的素数。

```
>>>
请输入开始值（整数）：275
请输入一个结束值（整数）：300
277 是素数。
281 是素数。
283 是素数。
293 是素数。
```

图 4.10 输出找到的素数

4.2.4 for 语句与内置迭代函数

Python 语言内置了几种常用的迭代函数，既方便又实用。

enumerate(seq)	#编号迭代
sorted(seq)	#排序迭代
reversed(seq)	#翻转迭代
zip(seq1,seq2,...)	#并行迭代

编号迭代在迭代时既返回序列中的元素，又返回该元素在序列中的编号（编号从 0 开始）。for 语句进行编号迭代时，应使用两个循环变量分别接收编号和元素的值。



在交互式环境下的示例代码以及输出如下：

```
>>> for i,item in enumerate('abcd'):  
    print('第%d个字符是:%s' % (i,item))
```

```
第0个字符是:a  
第1个字符是:b  
第2个字符是:c  
第3个字符是:d
```

排序迭代的作用是使得在 for 的遍历时先遍历序列中较小的值，后遍历序列中较大的值，当然这要求序列中的数据可以是排序的同类数据。

在交互式环境下使用排序迭代的示例代码如下：

```
>>> for i in sorted([3,1,6,0]):  
    print(i)
```

```
0  
1  
3  
6
```

从以上输出可以看出遍历依次得到的数值是从小到大排列的。

而翻转迭代就是将迭代序列中的元素从尾至头进行遍历，这里就不再举例进行说明了，你可以在交互式环境下尝试。

并行迭代也是一种很实用的迭代函数，它在遍历时同时遍历函数中给出的 seq1、seq2 等序列中同一序号的元素。

在交互式环境下运用并行迭代的示例代码如下：

```
>>> lsta = (1,2)  
>>> lstb = (3,4)  
>>> lstc = (5,6,7)  
>>> for i,j,k in zip(lsta,lstb,lstc):  
    print('%d:%d:%d' % (i,j,k))
```

```
1:3:5  
2:4:6
```



注意 当并行迭代函数中序列值的长度不一致时，只遍历到最短的序列的长度。

4.3 用 while 循环执行语句



for 语句以遍历对象的方式构造循环，的确给用户带来很大的方便，而有时却需要构造一种类似无限循环的程序控制结构或以某种不确定运行次数的循环，此时就可以使用本节介绍的 while 语句。

4.3.1 while 基础

while 也是 Python 语言中构造循环结构程序的语句之一，在 Python 语言中，虽然绝大多数循环结构都是用 for 语句来完成的，while 语句也可以完成 for 语句的功能，只不过不如 for 语句来得简单明了，而 while 语句在 Python 中主要用于构建特别的循环。本小节主要介绍的就是 while 语句。

while 语句的基本形式如下:

```
while <条件>:
    <语句 1>
else:
    <语句 2>                # 如果循环未被 break 终止, 则执行
```

与 for 循环不同的是, while 语句只有在测试条件为假时才会停止。在 while 语句的循环体中一定要包含改变测试条件的语句, 以保证循环能够结束, 避免死循环的出现。

while 语句包含与 if 语句相同的条件测试语句, 如果条件为真就执行循环体; 如果条件为假, 则终止循环。while 语句也有一个可选的 else 语句块, 它的作用与 for 循环中的 else 语句块一样, 当 while 循环不是由 break 语句终止的话, 则会执行 else 语句块中的语句。而 continue 语句也可以用于 while 循环中, 其作用是跳过 continue 后的语句, 提前进入下一个循环。

while 循环不像 for 循环可以遍历某一对象的集合, 用 while 语句构造循环语句, 最容易出现的问题就是测试条件永远为真, 导致死循环。因此在使用 while 循环时应仔细检查 while 语句的测试条件, 避免出现死循环。

【实例 4-11】演示了用 while 循环遍历输出列表, 实现本书实例 4-6 的功能, 代码如下:

```
alst = [1,2,3,4,5]
total = len(alst)           #获取列表中元素总数
i = 0                       #初始化循环控制变量
while i < total:            #i 从 0 至 total-1
    print(i,"的平方是: ",alst[i]*alst[i])
    i = i + 1               #修改循环控制变量
else:
    print('循环结束!')
```

【代码说明】由此可以看出, 用 while 语句去遍历列表等序列类型显然比 for 要写的代码多, 如果循环控制条件搞错的话很容易出现问题。所以一般情况下, 遍历序列类型都采用 for 语句, 而不用 while 语句。

【运行效果】见实例 4-6 后的图 4.6。

在 while 语句中使用 break 和 continue 语句的方法与效果和 for 语句相同。你可以对照本书实例 4-7, 并结合本书实例 4-11 改写出与本书实例 4-7 同样功能的代码。

4.3.2 增量赋值运算符

Python 中提供的增量赋值运算符很多, 基本的运算符都有对应的增量赋值运算符:

+, -, *, /, //, **, %, &, |, ^, >>, <<

基本写法形如:

```
x += 1
```

意义为:

```
x = x + 1
```

以上两种写法的运算含义相同, 但是增量赋值运算符书写更简单一点, 也很容易理解。



注意 增量运算符并不是赋值的结果比原来的值大。

增量运算符可以运用在任意使用普通赋值的地方, 但是要求增量赋值的参与赋值和运算的操作数能够适用相应的基础运算符。在 while 语句中, 用来增量运算符修改循环控制变量的值非常方便。比如本书实例 4-11 中的 while 循环体里的 “x = x + 1” 可写为 “x += 1”, 而 “x = 'abc';



`x+=3`”则会出现语法错误，因为字符串和整数是不能执行加法的。

4.4 推导或内涵 (list comprehension)

通过本书前面介绍可知，列表、元组、字典等结构数据类型给使用大量数据带来极大的方便，但可能你还需要对序列中的数据进行处理得到另一组数据序列，本节介绍从一个序列类型的数据集推导出另一个序列类型的数据集。

4.4.1 推导基础

`comprehension` 英文本义为理解或内涵（逻辑学用语），那么在 Python 中是指以紧凑的方式对列表、元组、字典等序列或一系列的元素进行处理，处理结果仍然被放到一个列表、字典等序列之中的语法形式。

典型的列表推导基本形式如下：

```
[ <i 相关表达式> for i in aiterator]
```

`aiterator` 是指一个可遍历的对象，比如列表、元组，也可以是 `range()` 函数。其语义为用循环变量 `i` 去遍历 `aiterator`，并且将 `i` 相关表达式的值放入一个列表中。

每当对列表、元组、字典序列中的元素进行处理时，你都应该尝试使用列表推导来完成，这样非常有助于降低程序的复杂性，即使得程序清晰易懂，也缩短了程序的长度。

比如以下两段代码都是获得 1~10 的平方数存入列表之中，使用列表推导只需要一行代码。列表推导代码如下：

```
square = [i*i for i in range(1,11)]
```

如果不使用列表推导，而使用 `for` 语句的代码则如下：

```
square = []
for i in range(1,11):
    square.append(i*i)
```

字典也是可以实现其推导式语法的，其基本格式如下：

```
{key_exp:value_exp for key_exp,value_exp in aiterator}
```

例如以下在交互式环境下执行的字典推导语句代码及输出：

```
>>> keys = ['name','age','weight']
>>> values = ['Bob','23','68']
>>> {k:v for k,v in zip(keys,values)}
{'weight': '68', 'age': '23', 'name': 'Bob'}
>>> adct = {k:v for k,v in zip(keys,values)}      #字典推导中使用了并行迭代
>>> adct
{'weight': '68', 'age': '23', 'name': 'Bob'}
```

4.4.2 推导进阶

列表推导式和字典推导式不仅可以对遍历的元素进行全部处理，还可以使用 `if` 语句实现有选择地处理遍历序列中的元素。其基本形式如下：

```
[ <i 相关表达式> for i in aiterator if <条件>]
{key_exp:value_exp for key_exp,value_exp in aiterator if <条件>}
```

如果要获得一个 1~10 中所有数的平方，且平方值为偶数的一个列表，就可以使用以下代码来实现：

```
square_odd = [i*i for i in range(1,11) if i*i %2 == 1]
```

对于字典推导式也可以使用 if 来进行部分元素的处理。

如果在推导式中的求值表达式或条件表达式中应用函数,则可以构造更加复杂的推导式来实现对序列中的数据进行批量的处理(这也常被称为声明式编程)。

4.5 小结

本章主要介绍了 Python 语言中控制语句执行流程的语法结构及其语义,首先介绍了构建分支结构程序的 if 语句,其次介绍了构建循环结构程序的 for 语句和 while 语句,最后介绍了推导。此外还介绍了与循环结构密切相关的 break、continue 语句、range()函数、内置迭代函数及增量赋值运算符。学习完本章后,你应该掌握用 Python 编写分支结构、循环结构的程序,并能熟练使用列表或字典推导,以及应用 Python 内置迭代函数来编程程序,解决实际问题。

4.6 本章习题

一、选择题

1. 有下面的程序段:

```
if k <= 10 and k > 0:
    if k>5:
        if k<8:
            x = 0
        else:
            x = 1
    else:
        if k > 2:
            x = 3
        else:
            x = 4
```

其中 k 取哪组值时 (), x 的值为 3。

- A.3, 4, 5
- B.3, 4
- C.5, 6, 7
- D.4, 5

【解析】阅读程序可知, k 必须在 0~10 之间、要小于等于 5, 且大于 2。答案为 A。

2. 有下面的程序段:

```
if k > 0:
    if j <= 0:
        pass #1
    elif j > 2:
        pass #2
    else:
        pass #3
pass #4
```

当 k=0, j=0 时, 执行的是 () 处的 pass 语句。

- A.#1
- B.#2
- C.#3
- D.#4

【解析】k>0 结果为假, 所以 if 内的语句块直接跳过, 而顺序向后执行。答案为 D。

3. 以下循环执行的次数是 ()。

```
for i in range(0,10,3):
    pass
```

- A.3
- B.4
- C.5
- D.6



【解析】起始为0，步进为3，所以i分别取得0、3、6、9共四次循环。答案为B。

4. 以下程序段输出的图形是（ ）。

```
for i in range(3):
    for j in range(i,3):
        print('+',end='')
    print()
```

A.

```
+
```

```
++
```

```
+++
```

B.

```
+++
```

```
++
```

```
+
```

C.

```
+
```

```
++
```

```
+++
```

D.

```
+++
```

```
++
```

```
+
```

【解析】外循环从0至2，内循环分别为0~2、1~2、2~2。答案为B。

二、实验题

1. 编程实现用户输入一门课程的两门子课程成绩，第一门子课程60分以上，则显示“通过”，第一门子课程不及格，则显示“未通过”，第一门子课程及格，而第二门子课程不及格，则显示“补考”。

2. 编程实现用户输入20个数，将所有数收集到一个列表中，然后将正数、负数放入两个列表并输出。

【提示】可以分别建立三个空列表，第一个列表用于收集所有数据，第二个列表收集正数，第三个列表收集负数。还可以直接使用列表推导来从第一个列表中按要求来取得符合条件的值。

3. 设有以下信息字符串：

```
myseq="""[a:1,b:2,c:3]
[a:3,b:3,c:8]
[a:7,c:2:m:7,r:4]
[a:2,c:4:m:6,r:4]
[a:3,b:2,c:7,o:5]"""
```

现要求去除其中重复的关键字，如上例的处理结果为：

```
'[a:1,b:2,c:3]', '[a:7,c:2:m:7,r:4]', '[a:3,b:2,c:7,o:5]'
```

【提示】建议尽量使用列表推导来处理，提取每个行中的键值，然后去除重复。

第 5 章 自定义功能单元（一）

Python 语言中已经内置了很多功能单元，其中最简单的功能单元就是函数。在 Python 语言的交互式环境下可以看到内建函数有不少，包括前面介绍的输入输出函数、数值类型转换函数、zip()、range()等迭代器函数。

那么，如果你想编程解决自己的实际问题，就需要自己定义相关函数并调用它来完成相关功能。

本章主要介绍 Python 中自定义函数及调用的方法，内容包括：

- 声明函数；
- 调用自定义函数；
- 变量作用域；
- 各种类型的函数参数应用；
- 使用 lambda 建立匿名函数；
- Python 其他常用内建函数。

5.1 使用函数



在编写程序的过程中，可以将完成重复工作的语句提取出来，将其编写为函数。这样，在程序中可以方便地调用函数来完成这些重复的工作，而不必重复地粘贴复制代码。此外，函数也可以使得程序结构更加清晰，更容易维护。

在 Python 中，函数必须先声明，然后才能调用它，使用函数时，只要按照函数定义的形式，向函数传递必需的参数，就可以调用函数完成相应的功能或者获得函数返回的处理结果。

5.1.1 声明函数

在 Python 中，使用 def 可以声明一个函数，完整的函数是由函数名、参数以及函数实现语句（函数体）组成的。同本书前面几章中所讲解的 Python 基本语句一样，在函数声明中，也要使用缩进以表示语句属于函数体。

如果函数有返回值，那么需要在函数中使用 return 语句返回计算结果，声明函数的一般形式如下：

```
def <函数名> (<参数列表>):  
    <函数语句>  
    return <返回值>
```

其中参数列表和返回值不是必须的，return 后也可以不跟返回值，甚至连 return 也没有。对于 return 后没有返回值的和没有 return 语句的函数都会返回 None 值。

有些函数可能既不需要传递参数，也没有返回值。



注意 没有参数时，包含参数的圆括号也必须写上，圆括号后也必须要有“:”。

下面以一个最简单的函数定义示例，相当于本书 1.5.1 小节中“你好，Python!”这个最简单程序的函数版，代码如下：



```
def hello():                # 函数名为hello, 无参数
    print('你好,Python!')   # 缩进的语句, 表示是函数内的语句
                             # 函数没有使用 return 定义返回值, 运行完会返回 None
```

以下是一个含有声明函数所有要素的函数声明, 实现了求一个元组中所有数之和的功能, 其参数 `T` 为所要求和的元组, `result` 就是元组求和的累加器, 最后函数使用 `return` 将累加的结果 `result` 返回。函数声明代码如下:

```
def tpl_sum( T ):
    result = 0
    for i in T:
        result += i
    return result
```

Python 的函数比较灵活, 与 C 语言中函数的声明相比, 在 Python 中声明一个函数不需要声明函数的返回值类型, 也不需要声明参数的类型。

5.1.2 调用函数

其实调用函数最先在本书的 2.2.1 小节中就已经使用了, 那里所介绍的输入函数(`input()`)和输出函数(`print()`)的使用时, 就是在调用 Python 的内建函数。而调用自己定义的函数与调用内建函数及标准库中的函数方法都是相同的, 要调用指定的函数就在语句中使用函数名, 并且在函数名之后用圆括号将调用参数括起来, 而多个参数之间则用逗号隔开。调用自定义函数与内建函数不同点在于自定义函数调用前必须先声明函数。



注意 如果调用函数的函数没有参数, 也必须在函数名后给出圆括号。

如果要想调用 5.1.1 小节定义的 `hello()` 函数时, 程序中应该先声明它, 然后就可以调用这个函数来运行函数版的“你好, Python!”, 完整的代码如以下实例。

【实例 5-1】 演示了函数版“你好, Python!”程序的声明与调用, 代码如下:

```
def hello():                #先声明, 声明时无参数也要写上括号
    print('你好,Python!')
hello()                     #后调用, 调用时无参数也要写上括号
```

【代码说明】 先声明 `hello()` 函数, 后调用 `hello()` 函数。

【运行效果】 见本书 1.5.2 小节图 1.19。

Python 在实际调用函数的过程中也非常得灵活, 不必为不同类型的参数声明多个函数, 在处理不同类型数据的时候可以调用相同的函数, 大部分情况都可以用同一个函数调用不同的数据类型。

参考 5.1.1 小节 `tpl_sum()` 函数可知, 当初定义的目的是求一个元组中所有数的和并返回的函数, 而函数声明中的参数并没有指定参数的类型。实际上, 调用该函数的时候, 不仅可以向其传递一个元组, 也可以向其传递一个列表, 同样可以对列表进行求和。通过阅读该函数的程序代码, 你会发现, 如果该函数的参数 `T` 为列表时, 函数体中的语句仍然是可以正常工作的。因此, 你在调用这个函数时, 提供的参数不管参数为一个列表, 还是一个元组, 函数都被正确的执行。

当然, 这并不表示可以向函数传递任何参数, 主要还是取决于函数的实现, 在 `tpl_sum()` 函数中, 只使用了 `for` 循环语句来遍历传入的参数, 并累加传入参数中的元素, 只要传入的参数可以遍历, `for` 循环就能够正确执行, 因此, 该函数才得以正确执行。但是, 如果调用时所给参数的元组或列表中的元素不是整数、浮点数或单纯的字符串, 则无法实现加法, 这就会导

致调用该函数时的运行失败，并且这也意味着一旦出现问题，只有在程序运行的时候才能被发现。

【实例 5-2】演示了 `tpl_sum()` 函数定义和调用的完整实例，代码如下：

```
def tpl_sum( T ):
    result = 0
    for i in T:
        result += i
    return result

print("(1,2,3,4)元组中元素的和为: ",tpl_sum((1,2,3,4)))
print("[1,2,3,4]列表中元素的和为: ",tpl_sum([1,2,3,4]))
print("[2.7,2,5.8]列表中元素的和为: ",tpl_sum([2.7,2,5.8]))
print("[1,2,'ab']列表中元素的和为: ",tpl_sum([1,2,'ab'])) #运行会出错
```

【代码说明】代码中先声明 `tpl_sum()` 函数，然后前两行函数调用分别提供了元组、列表两种实际参数，并输出函数的调用结果；第三次调用提供的列表参数中混合有整数和浮点数，第四次提供的列表参数中混合有整数和字符串。

【运行效果】由以上代码说明可知：其中的函数调用前三次是可以正常运行，并输出运算的结果；而第四次调用，因整数和字符串不能实现加法而运行失败，其运行结果如图 5.1 所示。

```
>>>
(1,2,3,4)元组中元素的和为: 10
[1,2,3,4]列表中元素的和为: 10
[2.7,2,5.8]列表中元素的和为: 10.5
Traceback (most recent call last):
  File "D:\lx\as_2.py", line 10, in <module>
    print("[1,2,'ab']列表中元素的和为: ",tpl_sum([1,2,'ab']))
  File "D:\lx\as_2.py", line 4, in tpl_sum
    result += i
TypeError: unsupported operand type(s) for +=: 'int' and 'str'
```

图 5.1 `tpl_sum()` 函数的调用结果

5.2 深入函数

在 Python 中，函数的参数除了上节介绍的一种方式之外，还可以有多种形式。例如在调用某些函数时，既可以向其传递参数，也可以不传递参数，函数依然可以正确调用；还有一些情况，比如函数中的参数数量不确定，可能为一个，也可能为几个、甚至几十个。对于这些函数，应该怎么定义其参数呢？

5.2.1 默认值参数

在 Python 中，可以在声明函数的时候，预先为参数设置一个默认值，当调用函数，如果某个参数具有默认值，则可以不向函数传递该参数，这时，函数将使用声明函数时为该参数设置的默认值来运行。

声明一个参数具有默认值的函数形式如下：

```
def <函数名> (参数=默认值):
    <语句>
```

【实例 5-3】声明了一个带默认值参数的函数，代码如下：

```
def hello(name='Python'):
    print('你好, %s!' % name)
```





```
print('无参数调用时的输出：')
hello()
print('有参数("Jonson")调用时的输出：')
hello('Jonson')
```

【代码说明】代码中声明的带默认值参数 `name` 的函数 `hello()`，若调用时不加参数，则 `name` 值被自动赋予 `"Python"` 字符串；若调用时加了参数，则 `name` 值会被赋予所给参数值。

【运行效果】如图 5.2 所示，第一次进行无参数调用，`name` 被赋予 `"Python"` 字符串，第二次调用时给了参数 `"Jonson"`。



```
>>>
无参数调用时的输出：
你好, Python!
有参数("Jonson")调用时的输出：
你好, Jonson!
```

图 5.2 默认值参数函数调用

如果在声明某一函数时，其参数列表中既包含无默认值参数又包含有默认值的参数，那么在声明函数的参数时，必须先声明无默认值参数，后声明有默认值参数。在交互式环境的实验示例及错误提示如下所示：

```
>>> def test(e=3,a):
    pass
SyntaxError: non-default argument follows default argument
```

以上错误提示说明：在声明函数 `test()` 时，参数列表中有具有默认值参数之后还有无具有默认值参数，因此发生了语法错误。

如果一个函数具有多个参数，而且这些参数都具有默认值，在调用函数的时候，可能仅想向最后一个参数传递值，该怎么办？

【实例 5-4】演示了带有两个具有默认值参数的函数，代码如下：

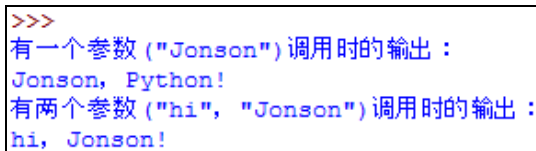
```
def hello(hi='你好',name='Python'):
    print('%s, %s!' % (hi,name))

print('有一个参数("Jonson")调用时的输出：')
hello('Jonson')

print('有两个参数("hi", "Jonson")调用时的输出：')
hello('hi','Jonson')
```

【代码说明】代码中两次调用 `hello` 函数，分别给出一个参数和两个参数进行调用。

【运行效果】如图 5.3 所示。



```
>>>
有一个参数("Jonson")调用时的输出：
Jonson, Python!
有两个参数("hi", "Jonson")调用时的输出：
hi, Jonson!
```

图 5.3 给出不同参数调用输出

从图 5.3 输出可以看出，在 Python 中传递参数是按照声明函数时定义的参数的顺序依次传递的。第一次调用时只给了一个参数 `"Jonson"`，结果是变量 `hi` 得到了它，`name` 变量没有得到并且被赋予了默认值。第二次调用时给了两个参数，变量 `hi`、`name` 按照顺序依次得到了调用时给的参数值。要想将调用时仅给的一个参数赋予第二个变量，就必须向指定的参数传递值，请

继续学习下节内容。

5.2.2 参数传递

在 Python 中参数值的传递是按照声明函数时参数的位置顺序进行传递的，即位置参数，调用时提供的第一个参数值会被声明时的参数列表中的第一个参数获取，其他的参数结合方法依次类推。而实际上 Python 还提供了另外一种传递参数的方法——按照参数名传递值的方法，即提供关键字参数，提供关键字参数的函数调用类似于定义函数时设置参数的默认值。

提供关键字参数调用函数时，要在调用函数名后的圆括号里写出形式为“关键字=参数值”的参数，这样，调用函数时提供参数就不必严格按照该函数声明时的参数列表的顺序来提供调用参数了。如果在函数调用时，既提供了关键字参数，又提供了位置参数，则位置参数会被优先使用，如果参数结合时出现重复，则会发生运行错误。

例如，对于实例 5-3 要只给第二个参数传递值，可以使用关键字参数的方式来调用函数 `hello()`，如下所示：

```
hello(name = "Jonson")
```

也可以使用以下代码给出两个参数来调用 `hello()`：

```
hello(name = "Jonson", hi="hi")
```

这种函数调用方法，虽然先给出了 `name`，后给出了 `hi`，但是使用了关键字参数的方式来调用的就可以达到目的。



注意 调用函数提供参数时，按顺序传递的参数要位于关键字参数之前，而且不能有重复的情况。

请看以下在交互式环境中，实验的代码其运行效果的示例，如图 5.4 所示。

```
>>> def mult_test(a,b,c):
      return a*b*c

>>> mult_test(2,c=5,b=3)
30
>>> mult_test(c=5,b=3,2)
SyntaxError: non-keyword arg after keyword arg
>>> mult_test(2,c=5,a=3)
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    mult_test(2,c=5,a=3)
TypeError: mult_test() got multiple values for argument 'a'
```

图 5.4 按位置和参数名传递参数示例

根据以上情况可知：一般来说，定义的函数如果既有无默认值的函数，又有有默认值的函数，应当把有默认值的参数放在后面。这样在调用函数时，前面的无默认值的参数就可以依次序取得调用时所给的值。对于默认值参数，没有依次取得时，就使用默认值，给函数调用带来很大的便利。

5.2.3 可变数量参数传递

在自定义函数时，如果参数名前加上一个星号“*”，则表示该参数就是一个可变长参数。在调用该函数时，如果依次序将所有的其他变量都赋值之后，剩下的参数将会收集在一个元组中，元组的名称就是前面带星号的参数名。

【实例 5-5】 演示了自定义一个只有一个带星号的参数的函数实例，代码如下：



```
def change_para_num(*tpl):
    print(type(tpl))          #输出 tpl 变量的类型
    print(tpl)

change_para_num(1)
change_para_num(1,2,3)
```

【代码说明】代码中定义了一个函数 `change_para_num(*tpl)`，它只有一个带星号参数的函数。

【运行效果】如图 5.5 所示，两次调用第一行输出类型为 `tuple`，即元组；第一次调用时只有一个参数，元组中也只有一个值；第二次调用时有三个参数，全部收集到了 `tpl` 变量中。

```
>>>
<class 'tuple'>
(1,)
<class 'tuple'>
(1, 2, 3)
```

图 5.5 收集参数

当自定义函数时，参数中含有前面所介绍的三种类型的参数，则一般来说带星号的参数应放在最后。当带星号的参数放在最前面时，仍然可以正常工作，但调用时后面的参数必须以关键字参数方式提供，否则会因其后的位置参数无法获取值而引发错误。

【实例 5-6】演示了一个同时有三种类型的参数的函数定义及调用，代码如下：

```
def change_para_num(*tpl,a,b=0):
    print('tpl:',tpl)
    print('a:',a)
    print('b:',b)

change_para_num(1,2,3,a=5)
change_para_num(1,2,3)          #该调用会出错，a 变量没有默认值，也不能获取值
```

【代码说明】代码中定义的函数 `change_para_num()` 有三种类型的参数，并且带星号的参数放在最前面。第一次调用时给出了三个位置参数和一个关键字参数，因此 `tpl` 变量收集了 1、2、3 作为一个元组，而变量 `a` 则取得值 5，变量 `b` 则使用了默认值；第二次调用没有提供关键字参数，无默认值的参数 `a` 没有获取到值，所以调用失败。

【运行效果】如图 5.6 所示，具有三种参数的函数调用，显然第一次调用成功，第二次调用失败。

```
>>>
tpl: (1, 2, 3)
a: 5
b: 0
Traceback (most recent call last):
  File "D:\lx\5_6.py", line 7, in <module>
    change_para_num(1,2,3)
TypeError: change_para_num() missing 1 required keyword-only
argument: 'a'
```

图 5.6 具有三种参数的函数调用

使用元组来收集参数的参数时，调用时提供的参数不能为关键字参数，如果要收集不定数量的关键字参数，可以在自定义函数时的参数前加两颗星，即 `**valuenam`，这样一来，多余的关键字参数就可以以字典的方式被收集到变量 `valuenam` 之中。



注意 收集关键字参数时要放在函数声明的参数列表中的所有参数之后。

【实例 5-7】演示了一个定义了收集关键字参数的示例，代码如下：

```
def change_para_dct(a,b=0,**adct):
    print('adct:',adct)
    print('a:',a)
    print('b:',b)

change_para_dct(1,k=3,b=2,c=3)
```

【代码说明】代码中定义了函数 `change_para_dct()`，其最后一个参数前有两颗星号，即为收集关键字参数到字典中。调用时，给出两个多余的关键字参数，运行时会被放入名称为 `adct` 的字典中。

【运行效果】如图 5.7 所示，显然多余的关键字参数 `k=3`、`c=3` 都被收集到字典 `adct` 中了。

```
adct: {'k': 3, 'c': 3}
a: 1
b: 2
```

图 5.7 收集关键字参数

这种收集字典的方式为函数中使用大量的默认值提供了方便，不用把大量的默认值全放在函数声明的参数中，而是把它放入程序中。

【实例 5-8】演示了一个带有大量默认参数的函数及其调用，代码如下：

```
def cube(name,**nature):
    all_nature = {'x':1,
                  'y':1,
                  'z':1,
                  'color':'white',
                  'weight':1}
    all_nature.update(nature)
    print(name,"立方体的属性:")
    print('体积:',all_nature['x']*all_nature['y']*all_nature['z'])
    print('颜色:',all_nature['color'])
    print('重量:',all_nature['weight'])

cube('first')
cube('second',y=3,color='red')
cube('third',z=2,color='green',weight=10)
```

#只给出必要参数的调用

#提供部分可选参数

#提供部分可选参数

【代码说明】代码中定义了一个函数 `cube()`，其参数为两个，第一个是普通参数 `name`，第二个为收集关键字参数。函数体中给了一个默认参数的字典，然后用字典的 `update()` 函数将调用时提供的关键字参数更新至默认参数的字典。在以后的应用中，直接从字典获取即可。最后用三种不同参数的方式调用 `cube` 函数。

【运行效果】三种调用方式下的输出如图 5.8 所示。

```
>>>
first 立方体的属性:
体积: 1
颜色: white
重量: 1
second 立方体的属性:
体积: 3
颜色: red
重量: 1
third 立方体的属性:
体积: 2
颜色: green
重量: 10
```

图 5.8 大量默认值的关键字参数



5.2.4 拆解序列的函数调用

前面使用函数调用时提供的参数都是位置参数和关键字参数，实际上调用函数时还可以把元组和字典进行拆解调用，如下所示。



- 拆解元组：提供位置参数；
- 拆解字典：提供关键字参数。

调用时使用拆解元组的方法是在调用时提供的参数前加一个*号；要拆解字典必须在提供的调用参数前加两个*号。

【实例 5-9】演示了拆解元组的函数调用，代码如下：

```
def mysum(a,b):  
    return a+b  
  
print('拆解元组调用：')  
print(mysum(*(3,4)))           #调用时拆解元组作为位置参数  
print('拆解字典调用：')  
print(mysum(**{'a':3,'b':4}))  #调用时拆解字典作为关键字参数
```

【运行效果】如图 5.9 所示，两种调用的结果是相同的。



图 5.9 拆解序列

5.2.5 函数调用时参数的传递方法

Python 中的元素有可变和不可变之分，如整数、浮点数、字符串、元组等都属于不可变的；而列表和字典都属于可变的。

第 3 章中对元组是不可变的已经介绍过了，列表和字典的可变也是很好理解的，即它们可以增减元素、修改元素的值。那么整数、浮点数、字符串等为什么是不可变的呢？本书的 3.3.2 小节介绍“=”时说，“=”号的作用是将对象引用与内存中某对象进行绑定，既然整数是不可变的，那么怎么改变一个指向整数的变量的值的呢？答案是直接在内存中创建一个新的整数值，然后将变量引用与其绑定，这在本质上虽与其他高级语言不同，但在使用上是看不出什么差别的，但若将其提供给函数作为参数，效果则不同。

在函数调用时，若提供的是不可变参数，那么在函数内部对其修改时，在函数外部其值是不变的；若提供是可变参数，则在函数内部对它修改时，在函数外部其值也会改变的。

【实例 5-10】演示了调用函数对提供的可变和不可变参数进行修改前后的效果，代码如下：

```
def change(aint,alst):           #定义函数  
    aint = 0                     #修改 aint 值  
    alst[0]=0                    #修改 alst 第一个值为 0  
    alst.append(4)               #在 alst 中添加一个元素： 4  
    print('函数中 aint:',aint)   #输出函数中的 aint 的值
```

```

print('函数中 alst:', alst)    #输出函数中的 alst 的值

aint = 3
alst = [1,2,3]
print('调用前 aint: ', aint)
print('调用前 alst: ', alst)
change(aint, alst)
print('调用后 aint: ', aint)    #调用后值和调用前值相同
print('调用后 alst: ', alst)   #调用后值和调用前值不同

```

【代码说明】代码中定义了一个修改提供参数的函数 `change()`，参数包括一个整数和一个列表。在调用前定义并输出了一个整数 `aint` 和一个列表 `alst`，然后把他们作为参数调用 `change()`，最后输出两个变量的值。

【运行效果】如图 5.10 所示，在调用前后，不可变量 `aint` 的值虽然在函数内部进行了改变，但函数退出后其值仍然不变；而可变的列表 `alst` 则完全不同，在调用函数后，显示了函数内的一切改变都应用了。但是在函数中，可变的和不可变的量的值的改变是应用的。

```

>>>
调用前 aint: 3
调用前 alst: [1, 2, 3]
函数中 aint: 0
函数中 alst: [0, 2, 3, 4]
调用后 aint: 3
调用后 alst: [0, 2, 3, 4]

```

图 5.10 函数对参数修改效果

列表、字典是可变对象，它在作为函数的默认参数时要注意一个“陷阱”。

【实例 5-11】演示了使用列表作为默认参数时出现的“陷阱”，代码如下：

```

def myfun(lst=[]):                #定义有一个默认值为空列表的参数
    lst.append('abc')
    print(lst)

myfun()                           #此后三次调用自定义函数
myfun()
myfun()

```

【代码说明】代码中只定义了一个带有空列表默认参数的函数 `myfun()`，然后在不提供参数的情况下调用了三次这个函数。

【运行效果】每次调用函数按默认值的理解，应该每次传入空列表，列表中只有一个元素 `'abc'`，但事实不是如此，结果如图 5.11 所示。

```

>>>
['abc']
['abc', 'abc']
['abc', 'abc', 'abc']

```

图 5.11 空列表默认参数调用结果

如果来实现空列表的默认参数，可以修改函数代码如下：

```

def myfun(lst=None):              #定义有一个默认值为空列表的参数
    lst = [] if lst is None else lst
    lst.append('abc')
    print(lst)

```



5.3 变量的作用域

在 Python 中，作用域可以分为以下这些。

- 内置作用域：Python 预先定义的；
- 全局作用域：所编写的整个程序；
- 局部作用域：某个函数内部范围。

每次执行函数，都会创建一个新的命名空间，这个新的命名空间就是局部作用域，同一函数在不同的时间运行，其作用域是独立的，不同的函数也可以具有相同的参数名，其作用域也是独立的。在函数内已经声明的变量名，在函数以外依然可以使用。并且在程序运行的过程中，其值并不相互影响。

【实例 5-12】演示了一个简单的程序，在函数内外都有同一个名称的变量而不影响，代码如下：

```
def myfun():
    a = 0                #函数内声明并初始化变量 a 为整数
    a += 3               #修改 a 的值
    print('函数内 a:',a) #输出函数内 a 的值

a = 'external'          #全局作用域内 a 声明并初始化

print('全局作用域 a:',a)
myfun()                  #调用函数 myfun()
print('全局作用域 a:',a)
```

【代码说明】代码在函数中声明了变量 `a`，其值为整数类型；在函数外声明了同名变量 `a`，其值为字符串。在调用函数前后，函数外声明的变量 `a` 的值不变。在函数内可以对 `a` 的值进行任意操作，它们互不影响。

【运行效果】不同作用域同名变量如图 5.12 所示，不同作用域下的同名变量互不影响。

```
>>>
全局作用域a: external
函数内a: 3
全局作用域a: external
```

图 5.12 不同作用域同名变量

上述实例中两个变量 `a` 处于不同的作用域中，所以互不影响，但是如 5.2.5 小节所述，如果将全局作用域中的变量作为函数的参数引用，则是不同的，但这两者不属于同一问题范畴。

但是，还有一种方法能使函数中引用全局变量并进行操作，如果要在函数中使用函数外的变量，可以在变量名前使用 `global` 关键字。

【实例 5-13】演示了使用 `global` 关键字，实现在函数内部使用全局变量的一种方式，代码如下：

```
def myfun():
    global a                #增加此语句
    a = 0
    a += 3
    print('函数内 a:',a)

a = 'external'
print('全局作用域 a:',a)
```

```
myfun()
print('全局作用域a:',a)
```

【代码说明】示例代码仅在函数 `myfun()` 声明中增加了一句，就使函数内使用的变量 `a` 成为全局变量。

【运行效果】如图 5.13 所示，函数改变了全局作用域变量 `a` 的值，即由字符串 ‘external’ 变为整数 3。

```
>>>
全局作用域a: external
函数内a: 3
全局作用域a: 3
```

图 5.13 函数内引用全局变量

在局部作用域内可以引用全局作用域内的变量，但不可以修改它。

比如以下代码是没有错误的：

```
a = 3                #定义全局变量
def myprint():       #声明函数 myprint()
    print(a)         #引用全局变量
```

运行函数 `myprint()` 时，会输出全局变量 `a` 的值 3。但若将其改为下述代码则会引发错误：

```
a = 3                #定义全局变量
def myprint():       #声明函数 myprint()
    print(a)         #引用全局变量
    a = 5
```

以上代码引发的错误是局部变量在分配前不能引用，原因与 Python 中的变量查找有关，在此外代码中，函数内声明了 `a` 变量并初始化，所以 `a` 被判为局部变量，但之前却在 `print(a)` 中引用了它。

5.4 使用匿名函数 (lambda)

`lambda` 可以用来创建匿名函数，也可以将匿名函数赋给一个变量供调用，它是 Python 中一类比较特殊的声明函数的方式，`lambda` 来源于 LISP 语言，其语法形式如下：

```
lambda params:expr
```

其中 `params` 相当于声明函数时的参数列表中用逗号分隔的参数，`expr` 是函数要返回值的表达式，而表达式中不能包含其他语句，也可以返回元组（要用括号），还允许在表达式中调用其他函数。

以下是在交互式环境下演示的 `lambda` 的示例代码：

```
>>> import math
>>> s = lambda x1,y1,x2,y2:math.sqrt((x1-x2)**2+(y1-y2)**2)
>>> s(1,1,0,0)
1.4142135623730951
```

代码的第二行定义了一个求两坐标点距离的匿名函数，并用 `s` 引用，之后调用它来求 (1, 1) 与 (0, 0) 坐标点的距离，结果为 1.414。

`lambda` 一般用来定义以下类型的函数。

- 简单匿名函数：写起来快速而简单，省代码；
- 不复用的函数：用于需要一个抽象简单的功能，又不想单独定义一个函数时；
- 为了代码清晰：有些地方使用它，代码更清晰易懂。



比如在某个函数中需要一个重复使用的表达式，就可以使用 `lambda` 来定义一个匿名函数，多次调用时，就可以少写代码，但条理清晰。

5.5 Python 常用内建函数

在 Python 中，没有导入任何模块或包时，Python 运行时提供的函数称为内建函数，除了前面介绍的函数以外，常用内建函数如表 5.1 所示。

表 5.1 常用内建函数

dir(obj)	列出对象的相关信息
help(obj)	显示对象的帮助信息
bin(aint)	十进制数转换为二进制数的字符串形式
hex(aint)	十进制数转换为十六进制数的字符串形式
oct(aint)	十进制数转换为八进制数的字符串形式
callable(obj)	测试对象是否可调用（函数）
chr(aint)	ascii 码转为字符
ord(char)	将字符转为 ascii 码
filter(seq)	对序列中的数据用函数过滤
map(seq)	对序列中的数据逐个变换
isinstance(obj, typestr)	测试对象是否为某类型

以下是在交互式环境下对使用这些函数的示例代码及其输出：

```
>>> help(abs)      #显示 abs 函数的帮助信息
Help on built-in function abs in module builtins:

abs(...)
    abs(number) -> number

    Return the absolute value of the argument.

>>> bin(20)                #十进制数 20 转换为二进制字符串形式
'0b10100'
>>> hex(20)                #十进制数 20 转换为十六进制字符串形式
'0x14'
>>> oct(20)                #十进制数 20 转换为八进制字符串形式
'0o24'
>>> callable(abs)         #检查 abs 是否可调用（函数）
True
>>> aa=3
>>> callable(aa)
False
>>> chr(97)                #ascii 码 97 转为字符
'a'
>>> ord('k')              #求字符的 ascii 码
107
>>> alst = [0,1,2,3,4]
>>> list(filter(lambda x:x % 2,alst))    #去除列表 alst 中偶数
[1, 3]
>>> list(map(lambda x:2*x,alst))        #将列表中 alst 中元素都扩大 2 倍
[0, 2, 4, 6, 8]
>>> isinstance('abc',str)    #检查'abc'是否是字符串类型
True
```

5.6 小结

5.7 本章习题

1. 调用以下函数返回的值为 ()。

【解析】Python 中函数体可以为空语句，无 return 语句时，返回 None。答案为 D。

【解析】从函数中的代码可知，函数的参数必须为可遍历的数据。答案为 C。

【解析】A 项参数的获取结果为 a=1,b=2。答案为 A。

【解析】函数调用时，提供的非关键字参数必须在关键字参数之前。答案为 D。

70



- A. 3 [1,2] 3 [1,2] B. 9 [1,2,3] 3 [1,2,3]
C. 9 [1,2,3] 9 [1,2,3] D. 9 [1,2,3] 3 [1,2,3]

【解析】函数参数传递时,全局的不可变量在函数内部的操作是不会改变的。答案为D。

6. 对于以下代码,运行后遍历 seq 得到的元素为()。

```
seq = [4,5,6,7,8,9]
seq = filter(lambda x:x%3 and x%2,seq)
```

- A. 4,6,8 B. 5,7
C. 5,6,8 D. 6,9

【解析】lambda 函数过滤了被3和2整除的数。答案为B。

二、实验题

1. 自定义函数实现对列表中的数值进行排序。
2. 自定义函数实现计算字符串中空格的个数。
3. 对于以下字符串信息:

```
<h3>联系我们</h3>
<p>联系人: 王经理</p>
<p>电话: 021-87017800</p>
<div id="nav">
  <ul>
    <li><a class="nav-first" href="/">首 页 </a></li>
    <li><a href="/lista.php">吸粮机</a></li>
    <li><a href="/listb.php">灌包机</a></li>
    <li><a href="/listc.php">汽油吸粮机</a></li>
    <li><a href="/order/setorder.php">我要订购</a></li>
    <li><a href="/about.php">关于我们</a></li>
  </ul>
</div>
```

请自定义函数获取所有的超链接。

【提示】使用超链接的标志性字符串 href=来搜索并获取。

4. 自定义函数实现对用户的登录验证,假设所有用户信息存放在列表中,形式如下:

```
[{'name': 'john', 'password': '2354kdd', 'usertype': 1},.....]
```

验证成功返回用户类型,否则返回0。

第 6 章 自定义功能单元（二）

前面章节中介绍的所有示例程序都属于面向过程的程序类型，Python 语言其实也是一种面向对象的编程语言。Python 中既可以用函数等面向过程的程序来解决相关的实际问题，也可以使用面向对象的方法完成相应的项目程序。因此，其灵活性是很强的。

面向对象是当今高级编程语言大多具有的特性，在学习面向对象编程前我们应该理解什么是程序中的对象、类的继承、面向对象的优点及怎样进行面向对象的编程等面向对象的编程的基本思想。其次，要学习在 Python 中如何进行面向对象的编程。

本章主要介绍 Python 语言面向对象编程的基本方法和基本思想，内容包括：

- 对象概述；
- 类与对象；
- 定义和使用类；
- 类的属性和方法；
- 类的继承；
- 类的方法重载。

6.1 面向对象编程概述

面向对象程序设计（Object Oriented Programming）简称 OOP，面向对象可以说是在计算机中通过编制程序的方式来模拟现实世界的物质运行方式的一种编程方法。通过面向对象编程，使得数据的管理更加合理和自动化，减少程序出错，少写代码，程序更加容易维护。

6.1.1 万物皆对象

在早期的面向过程的编程中，所有使用的数据和函数之间是没有任何直接联系的，它们之间联系的方式就是通过函数调用提供参数的形式将数据传入函数进行处理。聪明的程序员们发现这并不符合现实世界的运行规律，经常可能因为错误的传递参数、错误地修改了数据而导致程序出错，甚至是崩溃。当需要修改或维护程序时要从程序提供的一堆数据中去寻找和修改它，要扩展函数的功能，只能重新建立一个函数或修改它，所以其开发效率有点低。

而从现实世界得到的启发就是任何事物都具有自己的属性或能力，比如一张桌子有高度、材质、颜色、重量等属性；但它无生命，自己不能移动，也不具有完成其他动作的能力。再如一只小狗，也有毛色、重量、年龄、体重等属性；它有生命，可以自己走路、奔跑、叫唤等，具有自己的能力。

那么，在程序中可以模仿现实世界，对现实世界中的事物进行有目的抽象，即抽象出现实世界事物中对用户有用的属性和能力来建立一个关联在一起的模型，对于现实世界中事物没有的属性或能力，而程序中需要的，则可以在程序中进行添加；对于现实世界中事物具有的属性或能力，而程序中不需要关心的，则可以在程序中不进行表达。这种抽象出来的模型被称之为对象或类。

面向对象编程就是通过面向对象分析和设计，建立模型（类或对象）并完成最终程序的过程。因此，在面向对象编程中，编程的主体就是用类或对象构建模型，并使它们之间可以互相



通信以解决实际问题。

6.1.2 对象优越性

面向对象编程中的对象就是实际事物的模型或计算对象的模型,在程序中以类方式进行定义。类从某种意义上来说仍旧是对现实世界的模拟,它模拟的是现实世界中的各种事物,而现实世界中的各种事物都是具有类别的。比如阔叶植物、厨房用具、猫等名词所指的都是一类事物,在程序中定义的类就代表着同一类别的模型。

对象的优越性主要体现在以下几个方面:

- 封装;
- 继承;
- 包含。

封装是指将对象的属性和能力包装在一起,需要对外展示,其他对象才能得到或使用它,而不需要对外展示的细节则隐藏在对象的内部。比如一台洗衣机,它展示在用户面前的就是一些控制开关与按钮,人们就可以使用它,而其内部的电路结构、电机等就是隐藏的细节。对于外部的用户来说,不需要了解细节就可以通过按钮或开关来操纵它。同时,对象也使同一事物的属性与方法聚合在一起,而不能随便在别的什么地方定义。

继承是指通过获取父对象的属性及能力,再加上自定义的属性和能力而成为一个对象的子对象或一个类的子类。通过继承可以快速地对对象进行建模,进而节省大量时间去写已经存在的代码,还可以不失灵活性地修改父对象的某些特性(属性和能力)。

包含是指在对象建模的时候,还可以对对象模型进行细分。即将一个对象划分为几部分,分别进行建模,最后将它们组装在一起成为一个完整的对象。如对一张办公桌建模,可以先建立桌腿、桌面、抽屉等模型,最后组装在一起成为完美的办公桌。

6.1.3 类和对象

具有相同属性或能力的模型在面向对象编程中以类进行定义和表示的,由类可以派生出(实例化)出同类的各个实例。就像一枚印章一样,沾上不同颜色的墨水就可以印出不同颜色的文字或图形。仔细观察的话,即使使用了相同的墨水,每次印出的都应该属于不同的文字或图形个体,但其表达的意义相同。刻制一枚印章可以理解为定义一个类,印章的每一次使用都可以理解为类的一次实例化。

在 Python 语言中,前面所讲的数据类型其实也都是面向对象的。例如对于整数类型(int),每一个整数都是整数类的实例;对于浮点数类型,每一个实际的浮点数都是浮点数类的一个实例。

“对象”一词在面向对象编程中,根据上下文不同可以指“类”,也可以指“实例”,由此可见,它不是一个很准确的称呼。

6.2 定义和使用类

各种不同的面向对象程序设计语言定义类和使用类的方式大多是大同小异的,本节介绍 Python 语言中类的定义和使用方法。



6.2.1 定义类

在 Python 语言中定义类的基本形式为:

```
class <类名>(<父类名>):
    pass
```

- class 定义类的关键字；
- 类名 符合标志符规范的名称；
- 父类名 该类继承的父类名称；
- pass 空语句。

其中的父类名称是可选的，如果该类不继承其他类可以连同括号都不写；pass 语句表示什么也不做，常用来预留语句位置或临时未写等待以后完成，用一个单位缩进表示它属于这个类定义的一部分。



注意 class 语句行末尾要有一个“:”。

定义一个最简单的类只需要两行代码，如下：

```
class MyClass:
    pass
```

这个类表面上看什么都没有实现，也没有继承其他的类。但是在 Python 中，没有继承其他类的类，会自动继承系统中内建的类 object。下面就是在交互式环境下看到的 MyClass 类中从 object 类继承到的属性与方法，如图 6.1 所示。

```
>>> class MyClass:
>>>     pass
>>> dir(MyClass)
['_class_', '__delattr__', '__dict__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
 '__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__', '__weakref__']
```

图 6.1 默认从 object 继承来的属性与方法

6.2.2 使用类

类在定义后必须先实例化才能使用，类的实例化与函数调用类似，只要使用类名加圆括号的形式就可以实例化一个类。

类实例化以后会生成该类的一个实例，一个类可以实例化成多个实例，实例与实例之间并不会相互影响，类实例化以后就可以直接使用了。

【实例 6-1】 演示了一个简单类的定义与实例化，代码如下：

```
class MyClass:                                #定义一个类
    "MyClass help."                            #该类只有一个说明信息，没有具体语句

myclass = MyClass()                           #将自定义类 MyClass 实例化，名称为 myclass
print('输出类说明: ')
print(myclass.__doc__)                        #输出类实例 myclass 的属性 __doc__ 的值
print('显示类帮助信息: ')
help(myclass)                                #输出类的帮助信息
```

【代码说明】 代码中首先定义了一个自定义类 MyClass，其类体中只有一行类的说明信息，然后实例化该类，并调用类的属性来显示属性值。

【运行效果】 运行结果如图 6.1 所示。



```
>>>
输出类说明:
MyClass help.
显示类帮助信息:
Help on MyClass in module __main__ object:

class MyClass(builtins.object)
|   MyClass help.
|
|   Data descriptors defined here:
|
|   __dict__
|       dictionary for instance variables (if defined)
|
|   __weakref__
|       list of weak references to the object (if defined)
```

图 6.1 实例化类并显示相关信息

从以上类的帮助信息中还可以看到, MyClass 类自动继承了 builtins.object 类, 其后显示的属性就是从它继承而来的。

如果在类中定义了函数, 也可以用类实例来调用它。



注意

一般来说, 为了区分在类中定义的函数和类外定义的全局函数, 会将类中定义的函数称为方法。下文中出现的称呼便以此分类。

6.3 类的属性和方法



如上所述, 实例 6-1 所定义类 MyClass 只有一个说明信息, 是没有什么使用价值的。要用类来解决实际问题, 就要定义一个具有一些属性和方法的类, 因为这才符合真实世界中的事物特征。

6.3.1 类的方法

类的方法实际上是为类的能力建模的, 那么定义类的方法让类具有一定的能动性。在类外部调用该类的方法就可以完成相应的功能, 或改变类的状态或达到其他目的。

类中的方法定义和调用与函数定义和调用的方式基本相同, 其区别有:

- 方法的第一个参数必须是 self, 而且不能省略;
- 方法的调用需要实例化类, 并以实例名.方法名(参数列表)形式调用;
- 整体进行一个单位的缩进, 表示其属于类体中的内容。

【实例 6-2】演示了一个定义两个方法的类及其使用方法, 代码如下:

```
class Smp1Class:                                #定义一个类 Smp1Class

    def info(self):                             #定义一个类方法 info()
        print('我定义的类!')
```

```
    def mycacl(self,x,y):                       #定义一个类方法 mycacl()
        return x + y

sc = Smp1Class()                               #实例化类 Smp1Class()
print('调用 info 方法的结果: ')
sc.info()                                       #调用类实例 sc 的 info() 方法
print('调用 mycacl 方法的结果: ')
```

```
print(sc.mycacl(3,4))
```

```
#其中调用类实例 sc 的 mycacl()方法
```

【代码说明】代码中首先定义了一个具有两个方法 `info()` 和 `mycacl()` 的类，然后实例化该类，并调用其两个方法。

【运行效果】如图 6.2 所示，第一个方法调用直接输出信息，第二个方法调用计算了参数 3 与 4 的和。

```
>>>
调用info方法的结果:
我定义的类!
调用mycacl方法的结果:
7
```

图 6.2 方法调用演示



注意

定义方法时，也可以像定义函数一样声明各种形式的参数；方法调用时，不用提供 `self` 参数。

在 Python 语言中的类定义中，可以定义一个特殊的构造方法，即 `__init__()` 方法，用于在类实例化时初始化相关数据，如果在这个方法中有相关参数，则实例化时必须提供。



注意

在 `__init__()` 方法名中，`init` 前后分别为两个下划线。

【实例 6-3】演示了一个定义了构造方法的类，代码如下：

```
class DemoInit:

    def __init__(self,x,y=0):          #定义构造方法，具有两个初始化
        self.x = x
        self.y = y

    def mycacl(self):                  #定义应用初始化数据的方法
        return self.x + self.y

dia = DemoInit(3)                     #用一个参数实例化类
print('调用 mycacl 方法的结果 1: ')
print(dia.mycacl())

dib = DemoInit(3,7)                   #用两个参数实例化类
print('调用 mycacl 方法的结果 2: ')
print(dib.mycacl())
```

【代码说明】代码中定义了构造方法，此构造方法要求提供两个参数 `x`、`y`，其中 `y` 是具有默认值的参数。因此在实例化类时必须提供参数，根据函数的调用规则可知，实例化时至少提供一个参数，也可以提供两个参数。之后，分别以提供一个参数和两个参数的形式实例化类，并调用其 `mycacl()` 方法。

【运行效果】如图 6.3 所示，分别输出了两个类实例调用方法的结果。

```
>>>
调用mycacl方法的结果1:
3
调用mycacl方法的结果2:
10
```

图 6.3 构造方法的运行结果



类中的方法既可以调用本类中的方法,也可以调用全局函数来完成相关任务。调用全局函数和面向过程中的调用方式完成相同,而调用本类中的方法应用使用以下形式:

`self.方法名(参数列表)`

调用本类中的方法时,提供的参数列表中也不应该包含“self”。

【实例 6-4】演示了在类中调用类自身的方法和全局函数的实例,其代码如下:

```
def coord_chng(x,y):                #定义一个全局函数,模拟坐标值变换
    return (abs(x),abs(y))          #将 x,y 值求绝对值后返回

class Ant:                           #定义一个类 Ant

    def __init__(self,x=0,y=0):      #定义一个构造方法
        self.x = x
        self.y = y
        self.disp_point()           #构造函数中调用类中的方法 disp_point()

    def move(self,x,y):              #定义一个方法 move()
        x,y = coord_chng(x,y)       #调用全局函数,坐标变换
        self.edit_point(x,y)        #调用类中的方法 edit_point()
        self.disp_point()           #调用 类中的方法 disp_point()

    def edit_point(self,x,y):        #定义一个方法
        self.x += x
        self.y += y

    def disp_point(self):            #定义一个方法
        print("当前位置: (%d,%d)" % (self.x,self.y))

ant_a = Ant()                       #实例化 Ant()类
ant_a.move(2,4)                     #调用 ant_a 实例的方法 move()
ant_a.move(-9,6)                    #调用 ant_a 实例的方法 move()
```

【代码说明】代码中首先定义了一个全局函数 `coord_chng()`,然后定义了一个类,类中定义了一个构造方法,并且在构造方法中也调用了类中的其他方法(`disp_point()`)。此后定义的 `move()`方法同时调用了全局函数 `coord_chng()`和类中的两个方法(`edit_point()`和 `disp_point()`)。

【运行效果】如图 6.4 所示,由于初始化类 `Ant` 类时没有给出参数,所以使用了默认值(0,0);然后调用 `move()`方法,提供了参数 2,4,因此位置变为了 (2,4);第三次调用提供了参数-9,6,位置变为了 (11,10)。

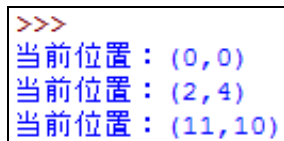


图 6.4 函数与方法调用示例

一般来说,为了方便程序代码的阅读、理解与维护,函数或方法的行数不要太多。你也看到实例 6-4 中有的方法只有一行或两行内容。

6.3.2 类的属性

类的属性是对类进行建模时必不可少的内容,方法是用来操作数据的,而操作的大部分数

据都类自身的属性，以改变类的状态，在 Python 中类的属性定义和使用方式是很方便的。

6.3.1 小节实例 6-4 的 Ant 类其实就已经定义了类的两个属性，它们是在构造方法中以直接赋值的方式定义的。因此，Python 中定义属性就是先直接使用它，可以在构造方法中定义属性，也可以在类中的其他方法使用定义属性。

Python 语言中类的属性有两类：

- 实例属性；
- 类属性。

实例属性即同一个类的不同实例，其值是互不关联的，也不会互相影响的，定义时使用“self.属性名”，调用时也使用它；类属性则是同一个类的所有实例所共有的，直接在类体中独立定义，引用时要使用“类名.类变量名”形式来引用，只要是某个实例对其进行修改，就会影响其他的所有这个类的实例。

【实例 6-5】演示了类中定义类属性和实例属性的定义和使用，代码如下：

```
class Demo_Property:                                #定义类
    class_name = "Demo_Property"                    #类属性

    def __init__(self,x=0):                           #实例属性
        self.x = x

    def class_info(self):                             #输出信息的方法
        print('类变量值: ',Demo_Property.class_name)
        print('实例变量值: ',self.x)

    def chng(self,x):                                #修改实例属性的方法
        self.x = x                                  #注意实例属性的引用方式

    def chng_cn(self,name):                          #修改类属性的方法
        Demo_Property.class_name = name             #注意类属性引用方式

dpa = Demo_Property()                               #实例化类
dpb = Demo_Property()                               #实例化类
print('初始化两个实例')
dpa.class_info()
dpb.class_info()
print('修改实例变量')
print('修改 dpa 实例变量')
dpa.chng(3)
dpa.class_info()
dpb.class_info()
print('修改 dpb 实例变量')
dpb.chng(10)
dpa.class_info()
dpb.class_info()
print('修改类变量')
print('修改 dpa 类变量')
dpa.chng_cn('dpa')
dpa.class_info()
dpb.class_info()
print('修改 dpb 实例变量')
dpb.chng_cn('dpb')
dpa.class_info()
dpb.class_info()
```

【代码说明】代码中首先定义了一个类 Demo_Property，类具有一个类属性 class_name 和



一个实例属性 `x`，以及两个分别修改实例属性和类属性的方法。其次分别实例化这个类，并用这两个类实例来修改类属性和实例属性。

【运行效果】如图 6.5 所示，对于实例属性来说，两个实例之间互不联系，它们各自可以被修改为不同的值；对于类属性来说，无论哪个实例修改了它，都会导致所有实例的类属性的值发生变化。

```
>>>
初始化两个实例
类变量值: Demo_Property
实例变量值: 0
类变量值: Demo_Property
实例变量值: 0
修改实例变量
修改 dpa 实例变量
类变量值: Demo_Property
实例变量值: 3
类变量值: Demo_Property
实例变量值: 0
修改 dpb 实例变量
类变量值: Demo_Property
实例变量值: 3
类变量值: Demo_Property
实例变量值: 10
修改类变量
修改 dpa 类变量
类变量值: dpa
实例变量值: 3
类变量值: dpa
实例变量值: 10
修改 dpb 实例变量
类变量值: dpb
实例变量值: 3
类变量值: dpb
实例变量值: 10
```

图 6.5 类属性和实例属性演示输出

有时为了不让某个属性或方法在类外被调用或修改，可以使用“`__`”双下画线的名称，但是这并不保证一定不能从类外调用，它只是一种标志。其实 Python 不提供类似其他面向对象语言中的私有属性和方法，你在编程时形成良好的习惯就可以了。

6.3.3 类成员方法与静态方法

类的属性有类属性和实例属性之分，类的方法也有不同的种类，主要有：

- 实例方法；
- 类方法；
- 静态方法。

前文定义的所有类中的方法都是实例方法，其隐含调用参数是类的实例。类方法隐含调用参数则是类，静态方法没有隐含调用参数。类方法和静态方法的定义方式都与实例方法不同，它们的调用方式也不同。

静态方法定义时应使用装饰器 `@staticmethod` 进行修饰，它是没有默认参数的。类方法定义时应使用装饰器 `@classmethod` 进行修饰，必须有默认参数“`cls`”。它们的调用方式可以直接由类名进行调用，调用前也可以不实例化类，当然也可以用该类的任一个实例来进行调用。

【实例 6-6】演示了一个同时定义了类方法和静态方法的类，代码如下：

```
class DemoMthd:                                #定义一个类

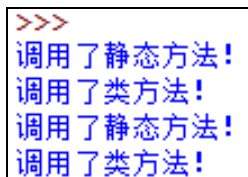
    @staticmethod                              #静态方法的装饰器
    def static_mthd():                          #静态类定义
        print('调用了静态方法! ')

    @classmethod                              #类方法的装饰器
    def class_mthd(cls):                       #类方法定义，带默认参数 cls
        print('调用了类方法! ')

DemoMthd.static_mthd()                        #未实例化类，通过类名调用静态方法
DemoMthd.class_mthd()                        #未实例化类，通过类名调用类方法
dm = DemoMthd()                              #实例化类
dm.static_mthd()                             #通过类实例调用静态方法
dm.class_mthd()                             #通过类实例调用类方法
```

【代码说明】代码中定义了一个同时具有静态方法和类方法的类 `DemoMthd`，然后在未实例化时用类名来调用它们；实例化后用类实例来调用它们。

【运行效果】如图 6.6 所示，两种方式调用都是正确的。



```
>>>
调用了静态方法!
调用了类方法!
调用了静态方法!
调用了类方法!
```

图 6.6 调用静态方法和类方法结果



注意

在静态方法和类方法中不能使用实例属性，因为有可能在调用时类还没有实例化。

6.4 类的继承

面向对象编程的最大优点之一就是可以通过继承来减少代码，同时也可以灵活定制新类。本节介绍的就是 Python 中类的继承。

6.4.1 类的继承

本书的 6.2.1 小节在类的定义形式中提到，类是可以继承的，并且也给出了继承类的代码格式。子类继承了父类之后，就具有了父类的属性与方法，但不能继承父类的私有属性和私有方法（属性名或方法名前缀为两个下画线的），子类中还可以重载来修改父类的方法，以实现与父类不同的行为表现或能力。

【实例 6-7】演示了类的继承，代码如下：

```
class Ant:                                     #定义类 Ant

    def __init__(self,x=0,y=0,color='black'):  #定义构造方法
        self.x = x
        self.y = y
        self.color = color

    def crawl(self,x,y):                       #定义方法（模拟爬行）
        self.x = x
```




```

        self.y = y
        print('爬行...')
        self.info()

    def info(self):
        print('当前位置: (%d,%d)' % (self.x,self.y))

    def attack(self):
        print("用嘴咬!")
#定义方法（模拟攻击）

class FlyAnt(Ant):
#定义 FlyAnt 类，继承 Ant 类

    def attack(self):
        print("用尾针!")
#修改行为（攻击方法不同）

    def fly(self,x,y):
        print('飞行...')
        self.x = x
        self.y = y
        self.info()
#定义方法（模拟飞行）

flyant = FlyAnt(color='red')
#实例化类
flyant.crawl(3,5)
#调用方法（模拟爬行）
flyant.fly(10,14)
#调用访求（模拟飞行）
flyant.attack()
#调用方法（模拟攻击）

```

【代码说明】代码中首先定义了父类 `Ant`，具有爬行和用嘴攻击能力，然后定义了继承 `Ant` 类的子类 `FlyAnt`，从 `Ant` 类中继承了爬行的能力，添加了飞行的能力，修改了攻击方式（由“用嘴咬”变成了“用尾针”）。之后实例化 `FlyAnt` 类，实例化调用其爬行、飞行及进攻方法。

【运行效果】如图 6.7 所示，父类具有爬行、攻击能力，子类添加了飞行能力，修改了进攻方式。

```

>>>
爬行...
当前位置: (3,5)
飞行...
当前位置: (10,14)
用尾针!

```

图 6.7 继承类的运行结果

6.4.2 多重继承

在面向对象编程的语言中，有的允许多重继承，即一个类可以继承多个类；有的只允许单一继承，即一个类只能继承一个类，而 `Python` 中则允许多重继承。

多重继承的方式是在类定义时的继承父类的括号中，以“,”分隔开要多重继承的父类。而在多重继承时，继承顺序也是一个很重要的要素，如果继承的多个父类中有相同的方法名，但在类中使用时未指定父类名，则 `Python` 解释器将从左至右搜索，即调用先继承的类中的同名方法。

【实例 6-8】演示了一个多重继承的实例，代码如下：

```

class PrntA:
#定义父类 PrntA

    namea = 'PrntA'

```

```

    def set_value(self,a):
        self.a = a

    def set_namea(self,namea):
        PrntA.namea = namea

    def info(self):
        print('PrntA:%s,%s' % (PrntA.namea,self.a))

class PrntB:                                # 定义父类 PrntB

    nameb = 'PrntB'

    def set_nameb(self,nameb):
        PrntA.nameb = nameb

    def info(self):
        print('PrntB:%s' % (PrntB.nameb,))

class Sub(PrntA,PrntB):                    # 定义子类 Sub, 先后继承了 PrntA,PrntB
    pass

class Sub2(PrntB,PrntA):                  # 定义子类 Sub, 先后继承了 PrntB ,PrntA
    pass

class Sub3(PrntA,PrntB):                  # 定义子类 Sub, 先后继承了 PrntA,PrntB

    def info(self):                        # 修改了方法 info
        PrntA.info(self)
        PrntB.info(self)

print('使用第一个子类: ')
sub = Sub()                               # 实例化类 Sub
sub.set_value('aaaa')
sub.info()
sub.set_nameb('BBBB')
sub.info()
print('使用第二个子类: ')
sub2= Sub2()                              # 实例化类 Sub2
sub2.set_value('aaaa')
sub2.info()
sub2.set_nameb('BBBB')
sub2.info()
print('使用第三个子类: ')
sub3= Sub3()                              # 实例化类 Sub3
sub3.set_value('aaaa')
sub.info()
sub3.set_nameb('BBBB')
sub.info()

```

【代码说明】代码中定义了两个父类 `PrntA` 和 `PrntB`，它们有一个同名的方法 `info()` 用于输出类的相关信息。第一个子类 `Sub` 先后继承了 `PrntA,PrntB`，实例化后，先调用了 `PrntA` 中的方法，之后调用了 `info()` 方法，由于两个父类中有同名的方法 `info()`，所以实际上调用了 `PrntA` 中的 `info()` 方法，因此只输出了从父类 `PrntA` 中的继承的相关信息。第二个子类 `Sub2` 继承的顺序相反，当调用 `info()` 方法时，实际上调用的是属于 `PrntB` 中的 `info()` 方法，因此只输出从父类 `PrntB` 中的继承的相关信息。第三个子类 `Sub3` 继承的类及顺序和第一个子类 `Sub` 相同，但是修改了父类的 `info()` 方法，在其中分别调用了两个父类的 `info()` 方法，因此，每次调用 `Sub3` 类实例的 `info()` 方法，两个被继承的父类中的信息都输出了。



【运行效果】如图 6.8 所示，使用第一、二个子类时，两次调用 info()方法仅输出了其中一个父类的信息。使用第三个子类时，每次调用 info()方法都同时输出了两个父类的信息。

```
>>>
使用第一个子类：
PrntA:PrntA,aaaa
PrntA:PrntA,aaaa
使用第二个子类：
PrntB:PrntB
PrntB:PrntB
使用第三个子类：
PrntA:PrntA,aaaa
PrntB:PrntB
PrntA:PrntA,aaaa
PrntB:PrntB
```

图 6.8 多重继承演示的运行结果

6.4.3 方法重载

当子类继承父类时，子类如果要想修改父类的行为，则应使用方法重载来实现，方法重载的基本方法是在子类中定义一个和所继承的父类中需要重载方法同名的一个方法。

例如在实例 6-7 中，子类 FlyAnt 继承了父类 Ant，父类中已经定义了方法 attack()，而子类中也定义了一个 attack()方法，即 attack()方法被重载了。当子类实例调用 attack()方法时，就会直接调用子类中的 attack()方法，而不会调用父类中同名的方法。

再如在实例 6-8 的多重继承中，两个父类都具有同名方法 info()，但在子类中也定义了一个 info()方法，即 info()方法被重载了。当子类实例调用 info()方法时，就会直接调用该实例中定义的 info()方法，而不会去调用任何一个父类的 info()方法。

此处不在另外举例说明。

6.5 小结

本章主要介绍了 Python 语言中面向对象编程的基本方法。首先介绍了面向对象的基本思想，其次介绍了在 Python 中如何定义和使用类、定义和使用类的属性与方法，最后介绍了类的继承与多重继承。通过本章的学习，应该重点掌握面向对象编程的基本思想，在 Python 中如何进行面向对象编程，以及定义和使用类、类的属性、类的各种方法、类的继承与方法重载。

6.6 本章习题

一、选择题

1. 面向对象编程的优越性不包括（ ）。

- A. 封装
- B. 继承
- C. 包含
- D. 高性能

【解析】与面向过程相比，面向对象的程序运行开销略大，所以性能也受影响。答案为 D。

2. 以下定义的类中，说法错误的是（ ）。

```
class TestCls:
    pass
```

- A. TestCls 类实例会包含__dir__方法
- B. TestCls 类实例会包含__repr__方法

- C. TestClss 类实例会包含__doc__方法
 D. 由于该类没有定义任何方法,所以其实例不包含任何方法

【解析】没有继承类的自定义类,会自动继承内建类 object,从而具有一些方法。答案为 D。

3. 以下定义的类中,说法错误的是()。

```
class TestClss:
    def prnt(sef):
        print(self.x)
```

- A. TestClss 类实例化会发生错误
 B. TestClss 类可以实例化
 C. TestClss 类的实例可以调用 prnt 方法
 D. TestClss 类实例化后直接调用 prnt 方法会出错

【解析】这个类是可以直接实例化的,但实例化后直接调用 prnt 方法会出错,因为类的实例属性 x 没有初始化,无法输出。答案为 A。

4. 以下程序运行输出的四个值分别为是()。

```
x = 3
class RefClss:
    def __init__(self,v=5):
        self.v = v
class TestClss:
    def __init__(self):
        self.ref = RefClss()
    def chng(self,x,y):
        self.info()
        x = 0
        self.ref.v = 0
        self.info()
    def info(self):
        print(x,self.v)
tc = TestClss()
tc.chng(0,0)
```

- A.3,5,3,0 B. 3,0,3,0
 C. 3,5,3,5 D. 3,0,0,0

【解析】面向对象的程序中、方法中、参数和函数调用中提供的参数作用相同。答案为 A。

5. 以下关于 Python 中类,说法错误的是()。

- A. 类的实例方法必须在类实例化后才能调用
 B. 类的实例方法可以在类实例化前调用
 C. 类的静态方法可以在类实例化前调用
 D. 类的类方法可以用类名或实例名调用

【解析】类的类方法和静态方法可以在实例化前调用,实例方法则不能。答案为 B。

二、实验题

1. 假设某游戏项目中需要定义一个精灵对象,其所需的属性有体重、颜色、高度、能量;具有行走、跳跃、进食能力,且会在行走和跳跃时会不断损耗能量,而进食则会增加能量。请根据描述定义这个精灵类。

【提示】应在构造方法中初始化其属性,并在行走和跳跃的方法中对能量值进行降低。

2. 根据本章中的 FlyAnt 和 Ant 类,自定义一个可以跳跃、爬行、飞行、用身子撞击的 FinalAnt 类。

第 7 章 错误、异常和程序调试

在 Python 编程中，常见的基本错误有两类，即语法错误和异常。对于语法错误，应该在程序编写过程中尽量避免，在程序调试中消除。而异常是 Python 程序在运行过程中引发的错误，如果在程序中引发了未进行处理的异常，程序就会由于异常而中止运行，只有为程序添加异常处理，才能使程序更“健壮”。

Python 对异常的处理，有它自己的语法形式。通过本章的学习，可以掌握如何在 Python 中处理异常和进行程序调试，本章主要内容有：

- 语法错误；
- 异常的概念；
- 用 try 语句捕获异常；
- 常见异常的处理；
- 自定义异常；
- 使用 pdb 调试 Python 程序。

7.1 语法错误

语法错误是所有编程语言中都存在的一种常见错误，即程序的写法不符合编程语言的规定。常见的语法错误有下面这些。



1. 拼写错误

即 Python 语言中的关键字被写错，变量名、函数名存在拼写错误等。

出现关键字拼写错误时系统会提示 `SyntaxError`（语法错误），而出现变量名、函数名拼写错误会在运行时给出 `NameError` 的错误提示，下面来进行讲解。

【实例 7-1】演示了一个语法上的错误，这里写错了一个单词，代码如下：

```
for i in range(3):  
    printt(i)
```

【代码说明】实例的代码第二行中的函数名 `print` 被错写成了 `printt`。

【运行效果】会出现 `NameError` 的错误提示，并同时指出错误所在的行等，如图 7.1 所示。

```
Traceback (most recent call last):  
  File "<pyshell#2>", line 2, in <module>  
    printt(i)  
NameError: name 'printt' is not defined  
>>>
```

图 7.1 语法错误提示

2. 脚本程序不符合 Python 的语法规范

例如少了括号、冒号等符号，以及表达式书写错误等。

3. 缩进错误

因为 Python 语法规则规定，以缩进作为程序的语法之一，这应该是 Python 语言独特的一面。一般来说 Python 标准的缩进是以四个空格作为一个缩进。当然，你可以依据自己的习惯，使用 Tab 键也可以。但同一个程序或项目中应该统一使用同一种缩进风格。

7.2 异常的处理

异常是 Python 程序在运行过程中引发的错误。如果程序中引发了未进行处理的异常，脚本就会由于异常而中止运行。只有在程序中捕获这些异常，并进行相关的处理，才能使程序不会中断运行。

7.2.1 异常处理的基本语法

Python 中使用 try 语句来处理异常，和 Python 中其他语句一样，try 语句也要使用缩进结构，try 语句也有一个可选的 else 语句块。一般的 try 语句基本形式如下：

```
try:
    <语句(块)>                #可能产生异常的语句(块)
except <异常名 1>:            #要处理的异常
    <语句(块)>                #异常处理语句
except <异常名 2>:            #要处理的异常
    <语句(块)>                #异常处理语句
.....
else:
    <语句(块)>                #未触发异常，则执行该语句(块)
finally:
    <语句(块)>                #始终执行该语，一般为了达到释放资源等目的
```

该语句的执行流程图，如图 7.2 所示。

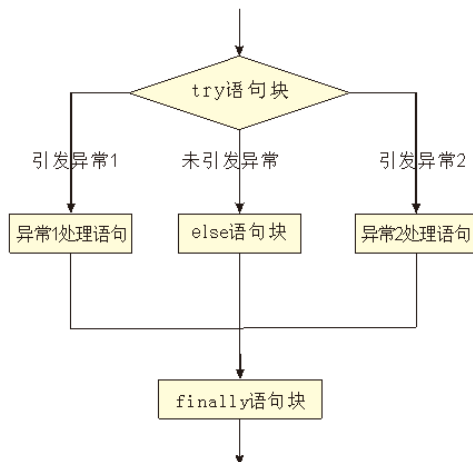


图 7.2 try 语句流程图



注意 else 语句块在未引发异常情况下得到运行。

在实际应用中可以根据程序的需要而使用部分语句，常见的形式有下面这些。
形式一：



```
try:
    <语句(块)>
except <异常名1>:
    <语句(块)>
```

【实例 7-2】演示了对于没有运行时异常的程序，不论是否捕获异常，程序都会正常运行；但是如果程序中发生了运行时异常，被捕获则程序运行不会中断，否则程序运行会中断退出。

```
def testTry(index,flag=False):
    stulst = ["John","Jenny","Tom"]
    if flag:                                     #flag 为 True 时，捕获异常
        try:
            astu = stulst[index]
        except IndexError:
            print("IndexError")
        return "Try Test Finished!"
    else:                                       #flag 为 False 时，不捕获异常
        astu =stulst[index]
        return "No Try Test Finished!"
print("Right params testing start...")
print(testTry(1,True))                        #不越界参数，捕获异常
print(testTry(1,False))                      #不越界参数，不捕获异常
print("Error params testing start...")
print(testTry(4,True))                        #越界参数，捕获异常
print(testTry(4,False))                      #越界参数，不捕获异常
```

【代码说明】本例定义了一个用于测试捕获异常的函数 testTry。flag 为 True 时，函数 testTry 运行时捕获异常，反之，该函数运行时不捕获异常。当传入的 index 参数正确时（不越界），测试结果都是正常运行的。当传入 index 错误（越界）时，如果不捕获异常，则程序运行中断。

【运行效果】如图 7.3 所示。

```
Right params testing start...
Try Test Finished!
No Try Test Finished!
Error params testing start...
IndexError
Try Test Finished!
Traceback (most recent call last):
  File "D:\lx\mytest.py", line 17, in <module>
    print(testTry(4))
  File "D:\lx\mytest.py", line 10, in testTry
    astu =stulst[index]
IndexError: list index out of range
```

图 7.3 异常的引发与捕获

形式二：

```
try:
    <语句(块)>
except <异常名1>:
    <语句(块)>
finally:
    <语句(块)>
```

【实例 7-3】演示了运用 finally 语句块来确保文件使用后能关闭该文件，代码如下：

```
def testTryFinally(index):
```

```

stulst = ["John", "Jenny", "Tom"]
af = open("my.txt", 'wt+')
try:
    af.write(stulst[index])
except:
    pass
finally:
    af.close()
    print("File already had been closed!")
print('No IndexError...')
testTryFinally (1)
print('IndexError...')
testTryFinally (4)

```

无论是否产生越界异常，都关闭文件

无越界异常，关闭文件

有越界异常，级关闭文件

【代码说明】本实例定义了一个测试函数 `testTryFinally`，在异常捕获代码中加入了 `finally` 代码块，其中的代码块用来关闭文件，并输出一行信息。无论传入的 `index` 参数值是否导致发生运行时异常（越界），总是能正常关闭打开的 `my.txt` 文件。

【运行效果】程序代码运行的效果如图 7.4 所示。

```

No IndexError...
File already had been closed!
IndexError...
File already had been closed!

```

图 7.4 finally 运行效果

7.2.2 Python 主要的内置异常及其处理

在 Python 中，常见的异常都已经预定义好了，在交互式环境中，用 `dir(__builtins__)` 命令就会显示出所有的预定义异常。常用的预定异常名及其描述如表 7.1 所示。

表 7.1 常用的异常名及其描述

异常名	描述
<code>AttributeError</code>	调用不存在的方法引发的异常
<code>EOFError</code>	遇到文件末尾引发的异常
<code>ImportError</code>	导入模块出错引发的异常
<code>IndexError</code>	列表越界引发的异常
<code>IOError</code>	I/O 操作引发的异常，如打开文件出错等
<code>KeyError</code>	使用字典中不存在的关键字引发的异常
<code>NameError</code>	使用不存在的变量名引发的异常
<code>TabError</code>	语句块缩进不正确引发的异常
<code>ValueError</code>	搜索列表中不存在的值引发的异常
<code>ZeroDivisionError</code>	除数为零引发的异常

`except` 语句主要有以下几种用法。

- `except`: # 捕获所有异常;
- `except <异常名>`: # 捕获指定异常;
- `except (异常名 1, 异常名 2)`: # 捕获异常名 1 或者异常名 2;
- `except <异常名> as <数据>`: # 捕获指定异常及其附加的数据;
- `except (异常名 1, 异常名 2) as <数据>`: # 捕获异常名 1 或者异常名 2 及异常的附加数据。



【实例 7-4】演示了程序中若捕获所有异常，则无论运行引发了什么异常，程序都不会中断。

```
def testTryAll(index,i):
    stulst = ["John","Jenny","Tom"]
    try:
        print(len(stulst[index])/i)
    except:                                #捕获所有异常
        print("Error!")

print('Try all...Right')
testTryAll(1,2)                          #正常输出结果
print('Try all...one Error')
testTryAll(1,0)                          #发生除 0 异常
print('Try all...two Error')
testTryAll(4,0)                          #越界异常和除 0 异常同时发生
```

【代码说明】代码中定义了函数 testTryAll，try 语句中捕获了所有的异常。第三次测试中，虽然同时发生了越界异常和除 0 异常，但程序不会中断，因为 try 语句中的 except 捕获了所有异常。

【运行效果】该实例运行效果如图 7.5 所示。

```
Try all...Right
2.5
Try all...one Error
Error!
Try all...two Error
Error!
```

图 7.5 捕获全部异常

【实例 7-5】演示了程序中捕获了部分异常，当程序运行时引发了不能被捕获的异常时，仍然会中断，代码如下：

```
def testTryOne(index,i):
    stulst = ["John","Jenny","Tom"]
    try:
        print(len(stulst[index])/i)
    except IndexError:
        print("Error!")

print('Try one...Right')
testTryOne(1,2)                          #正常输出结果
print('Try one...one Error')
testTryOne(4,2)                          #发生越界异常被捕获，程序不会中断
print('Try one...one Error')
testTryOne(1,0)                          #发生除 0 异常，未捕获，程序中断，有错误提示
```

【代码说明】代码中定义了函数 testTryOne，其 try 语句中只捕获了指明的 IndexError 异常，因此，当程序引发了 IndexError 异常时，程序不会中断。而当程序引发了除 0 异常时，程序会中断运行。

【运行效果】程序的运行效果如图 7.6 所示。

```
>>>
Try one...Right
2.5
Try one...one Error
Error!
Try one...one Error
Traceback (most recent call last):
  File "D:\lx\C7\7_5.py", line 13, in <module>
    testTryOne(1,0)
  File "D:\lx\C7\7_5.py", line 4, in testTryOne
    print(len(stulst[index])/i)
ZeroDivisionError: division by zero
```

图 7.6 捕获部分异常

由此可以看出,捕获所有异常,则出现任何错误都不会使程序中断。但若同时捕获所有异常,有时会使程序出现异常时程序员不知所措,找不到问题所在。



注意 一般情况下,应在程序中指明所要捕获的异常,而不是简单地捕获所有异常。

此外,异常处理的 try 语句也是可以嵌套的。

7.3 手工抛出异常



在上一节程序中,所有的异常都是在程序运行中出现了错误而引发的异常,程序员还可以在 Python 程序中使用 raise 语句来引发指定的异常,并向异常传递数据。

根据程序的需要,程序员还可以自定义新的异常类型,例如对用户输入文本的长度有要求,则可以使用 raise 引发异常,以确保文本输入的长度符合要求。

7.3.1 用 raise 手工抛出异常

使用 raise 引发异常十分简单,raise 有以下几种使用方式。

```
raise 异常名
raise 异常名,附加数据
raise 类名
```

使用 raise 可以抛出各种预定的异常,即使程序在运行时根本不会引发该异常。

【实例 7-6】演示了程序中使用了代码抛出异常,因为没有捕获该异常,所以程序运行会中断,导致后面的代码不能运行,如下:

```
def testRaise():
    for i in range(5):
        if i==2:
            raise NameError
        print(i)
    print('end...')

testRaise()
```

【代码说明】演示了代码中定义了函数 testRaise,函数中是一个 for 循环,当循环变量 i 为 2 时,抛出 NameError 异常,因没有处理该异常而导致程序运行中断,后面的所有输出都得不到执行。



【运行效果】程序运行的效果如图 7.7 所示。

```
>>>
0
1
Traceback (most recent call last):
  File "D:\lx\C7\a7_6.py", line 8, in <module>
    testRaise()
  File "D:\lx\C7\a7_6.py", line 4, in testRaise
    raise NameError
NameError
```

图 7.7 未捕获异常

【实例 7-7】演示了程序中使用了代码抛出异常，同时捕获了该异常，因此程序运行不会中断，代码如下：

```
def testRaise2():
    for i in range(5):
        try:
            if i==2:
                raise NameError
        except NameError:
            print('Raise a NameError!')
            print(i)
            print('end...')

testRaise2()
```

【代码说明】代码中定义了函数 testRaise2，函数中是一个 for 循环，当循环变量 i 为 2 时，抛出 NameError 异常，但是这个异常引发会被捕获处理，程序就不会中断，后面的所有输出都得到执行。

【运行效果】程序的运行效果如图 7.8 所示。

```
>>>
0
1
Raise a NameError!
2
3
4
end...
```

图 7.8 异常被捕获

7.3.2 assert 语句

assert 语句的一般形式如下。

assert <条件测试>,<异常附加数据> # 其中异常附加数据是可选的

assert 语句是简化的 raise 语句，它引发异常的前提是其后面的条件测试为假。

【实例 7-8】演示了程序中使用了 assert 抛出异常，同时捕获了该异常，代码如下：

```
def testAssert():
    for i in range(3):
        try:
            assert i<2
```

```

except AssertionError:
    print('Raise a AssertionError!')
    print(i)
    print('end...')

testAsser()

```

【代码说明】代码中定义了函数 testAsser，函数中是一个 for 循环，当循环变量 i 为 2 时，assert 后的条件测试会变为假，抛出 AssertionError 异常，但是这个异常引发会被捕获处理，程序不会中断，后面的所有输出都得到执行。

【运行效果】程序的运行效果如图 7.9 所示。

```

0
1
Raise a AssertionError!
2
end...

```

图 7.9 assert 运行效果

assert 语句一般用于在程序开发时测试代码的有效性。比如某个变量的值必须在一定范围内，而运行时得到的值不符合要求，则引发该异常，对开发者予以提示。所以一般在程序开发中不去捕获这个异常，而是让它中断程序。原因是程序中已经出现了问题，不应继续运行。

assert 语句并不是总是运行的，只有 Python 内置的一个特殊变量 __debug__ 为 True 时才运行。要关闭程序中的 assert 语句就使用 python -O（短画线，后接大写字母 O）来运行程序。如实例 7-8 用这个方法运行的结果如图 7.10 所示，很明显其中的 assert 语句被关闭，没有运行并引发异常。

```

D:\lx>python -O 6-8.py
0
1
2
end...

```

图 7.10 命令行运行关闭 assert

7.3.3 自定义异常类

在 Python 中定义异常类不用从基础完全自己定义，只要通过继承 Exception 类来创建自己的异常类。异常类的定义和其他类没有区别，最简单的自定义异常类甚至可以只继承 Exception 类，类体为 pass：

```

class MyError(Exception):          # 继承 Exception 类
    pass

```

如果需要异常类带有一定的提示信息，也可以重载 __init__ 和 __str__ 这两个方法。

【实例 7-9】演示了程序自定义了一个异常类，并用代码引发异常，代码如下：

```

class RangeError(Exception):

    def __init__(self,value):
        self.value = value

    def __str__(self):

```



```
return self.value

raise RangeError('Range Error!')
```

【代码说明】代码中自定义了一个继承了 `Exception` 类的异常类，并重载了 `__init__` 和 `__str__` 两个方法。之后，直接用 `raise` 抛出这个自定义的异常。

【运行效果】程序运行效果如图 7.11 所示。

```
>>>
Traceback (most recent call last):
  File "D:/lx/6-9.py", line 9, in <module>
    raise RangeError('Range Error!')
RangeError: Range Error!
```

图 7.11 抛出自定义异常

7.4 用 pdb 调试程序

Python 解释器可以发现程序中的语法错误，在试运行时会中断执行并给出提示。但是程序中逻辑上的错误，或其他非语法错误不会被发现。虽然程序能够正常运行，但是运行后得不到预想的结果，这时就要对程序进行调试。

调试程序可以使用 Python 自带的 `pdb` 模块，其功能有设置断点、单步执行、查看变量等。它可以用命令行参数的形式启动，也可以通过导入模块使用。常用的 `pdb` 模块的函数可以分为下面这几类。

7.4.1 调试语句块函数

`pdb` 模块中的调试语句块的函数及参数原型为：

```
run( statement[, globals[, locals]])
```

- `statement` 要调试的语句块，以字符串的形式表示；
- `globals` 可选参数，设置 `statement` 运行的全局环境变量；
- `locals` 可选参数，设置 `statement` 运行的局部环境变量。

【实例 7-10】交互模式下 `pdb` 模块调试，代码如下：

```
import pdb
pdb.run("""
for i in range(3):
    print(i)
""")
```

【代码说明】代码中首先导入 `pdb` 模块，调用 `pdb` 模块的 `run` 函数来调试一段简单的 Python 代码（字符串形式）。

【运行效果】如图 7.12 所示，在交互模式下调试。其中“(Pdb)”是 `pdb` 调试的提示符。在提示符下使用 `(help)` 命令可以查看所有的调试命令（如图中所示）。`n` 命令表示执行下一句；`continue` 命令表示继续执行以后的程序段。此外，`print(r)` 命令可用于输出变量的当前值，其他命令参考表 7.2。

`<string>(3)<module>()` 则表示即将执行代码语句行数和所在模块为匿名模块。

```

> <string>(2)<module>()
(Pdb) h

Documented commands (type help <topic>):
=====
EOF      c          d          h          list       q          rv          undisplay
a        cl       debug    help      ll         quit      s          unt
alias    clear    disable  ignore    longlist  r         source    until
args     commands display  interact  n         restart  step      up
b        condition down    j         next      return    tbreak   w
break    cont     enable  jump     p         retval   u         whatis
bt       continue exit    l         pp        run      unalias  where

Miscellaneous help topics:
=====
pdb      exec

(Pdb) n
> <string>(3)<module>()
(Pdb) continue
0
1
2

```

图 7.12 pdb 调试示例及命令演示

表 7.2 pdb交互命令表

完整命令	简写命令	描 述
args	a	打印当前函数的参数
clear	cl	清除断点
break	b	设置断点
condition	无	设置条件断点
continue	c 或者 cont	继续运行，直到遇到断点或者程序结束
disable	无	禁用断点
enable	无	启用断点
help	h	查看 pdb 帮助
ignore	无	忽略断点
jump	j	跳转到指定行数运行
list	l	列出程序清单
next	n	执行下条语句，遇到函数不进入其内部
p	p	打印变量值，也可以用 print
quit	q	退出 pdb
return	r	一直运行到函数返回
tbreak	无	设置临时断点，断点只中断一次
step	s	执行下一条语句，遇到函数进入其内部
where	w	查看所在的位置
!	无	在 pdb 中执行语句

7.4.2 调试函数

pdb 模块中的调试函数应当调用模块中的 `runcall` 函数，其参数原型为：

```
runcall( function[, argument, ...])
```

- `function` 函数名；
- `argument` 函数的参数。



【实例 7-11】演示了交互模式下 pdb 模块调试函数实例，代码如下：

```
import pdb

def sum(maxint):
    s = 0
    for i in range(maxint):
        s += i
    return s

pdb.runcall(sum,10)
```

【代码说明】代码中首先导入 pdb 模块, 调用 pdb 模块的 runcall 函数来调试自定义函数 sum, 然后调用 pdb.runcall 对其进行交互式调试。

【运行效果】如图 7.13 所示，提示符中指明了运行的文件、函数及当前语句行内容，可以多次调用并调试。

```
> d:\lx\6-11.py(4) sum()
-> s = 0
(Pdb) n
> d:\lx\6-11.py(5) sum()
-> for i in range(maxint):
(Pdb) n
> d:\lx\6-11.py(6) sum()
-> s += i
(Pdb) n
> d:\lx\6-11.py(5) sum()
-> for i in range(maxint):
(Pdb) continue
>>> pdb.runcall(sum,10)
> d:\lx\6-11.py(4) sum()
-> s = 0
(Pdb) continue
45
```

图 7.13 调试函数运行效果图

此外，pdb 模块还有 `runeval` 函数，可以用来调试表达式，读者可自行参考相关材料了解。

7.5 测试程序

编写完程序并排除了语法错误之后，虽然程序可以运行，但程序的运行结果和期待的可能会不一致。这说明程序中可能有 **bug**，即程序的逻辑错误。想发现和清除这类错误就要对程序进行测试，最好的方法就是使用测试驱动的开发（TDD）。对程序进行测试的种类包括：可用性测试、功能测试、单元测试及整合测试等。

Python 标准库中, 就有 `doctest` 和 `unittest` 模块可用于测试。本节主要讲解采用 Python 标准模块 `doctest` 中的 `testmod` 函数和 `testfile` 函数进行简单的单元测试。

7.5.1 用 testmod 函数测试

用 `testmod` 函数进行单元测试，就要将测试用例写入程序的 `docstring` 中，然后可以用两种方法进行测试。

【实例 7-12】演示了使用 doctest 模块的 testmod 函数进行基本的单元测试，代码如下：

```
import pdb
def grade(sum):
    """
    """
    >>> grade(100)
```

```

    '优秀'
>>> grade(80)
    '良'
>>> grade(65)
    '合格'
>>> grade(10)
    '不合格'
    """
    if sum > 90:
        return '优秀'
    if sum > 80:
        return '良'
    if sum > 60:
        return '合格'
    if sum < 60:
        return '不合格'

if name == '__main__':
    import doctest
    doctest.testmod()

```

【代码说明】代码中定义了一个根据考试分数返回考试评价的函数，并在其中的 docstring 中加入了测试用例，其中形如 “>>> grade(100)” 就是一个测试用例，并在下一行写出期待返回的结果。程序中共写入四个测试用例。



注意 >>> grade(100)语句中的>>>后要有一个空格。

【运行效果】如图 7.14 所示，测试结果信息共运行了四个测试用例，有一个测试失败并且指出了是哪个测试失败，测试期待返回值和真实返回值。

```

*****
File "D:/lx/7-10.py", line 5, in __main__.grade
Failed example:
    grade(80)
Expected:
    '良'
Got:
    '合格'
*****
1 items had failures:
  1 of  4 in __main__.grade
***Test Failed*** 1 failures.

```

图 7.14 测试信息

这种写法就是当单独运行这个程序时，就会对其进行测试。如果不希望这样，还可以通过在命令行下运行以下命令进行测试：

```
python -m doctest 7-14.py
```

其运行结果信息应该和图 7.14 的测试信息一致。

7.5.2 用 testfile 函数测试

上一节中的测试函数要求测试用例是写在程序文件的 docstring 中的。如果因为某种原因，不想或不能将测试用例写入程序文件中，就可以应用 testfile 函数进行测试。

【实例 7-13】演示了使用 doctest 模块的 testfile 函数进行基本的单元测试，代码如下：

```
import pdb
```




```
def grade(sum):  
    if sum > 90:  
        return '优秀'  
    if sum > 80:  
        return '良'  
    if sum > 60:  
        return '合格'  
    if sum < 60:  
        return '不合格'
```

将测试用例写入一个 mytest.txt 文件之中，内容如下：

```
from test import grade  #将上面程序文件存入 test.py 文件中  
>>> grade(100)  
'优秀'  
>>> grade(80)  
'良'  
>>> grade(65)  
'合格'  
>>> grade(10)  
'不合格'
```

【代码说明】代码中定义了一个根据考试分数返回考试评价的函数。并在 mytest.txt 文件中写入了四个测试用例。

如果在交互模式下运行，可使用以下程序命令：

```
import doctest  
doctest.testfile('mytest.txt')
```

如果在命令行模式下运行测试，可使用命令：

```
python -m doctest mytest.txt  
其运行效果和实例 7-12 完全一致。
```

7.6 小结

本章主要介绍在 Python 语言中的错误、异常和程序的简单调试。首先介绍了基本语法错误、异常的基本概念，以及如何用代码捕获和引发异常及自定义异常的方法。其次介绍了使用 Python 的内置模块 pdb 调试 Python 程序。最后讲解运用标准库 doctest 进行简单的单元测试。

通过对本章的学习，读者能够掌握引发异常、捕获和处理异常及自定义异常；会使用 pdb 模块进行基本的程序调试；能用 doctest 模块对程序进行基本的单元测试。

7.7 本章习题

一、选择题

1. 包含以下哪项的 Python 程序能启动运行（ ）。

- A. 拼写错误
- B. 错误表达式
- C. 缩进错误
- D. 手工抛出异常语句

【解析】手工抛出异常的语句会在运行时抛出异常，但不会影响程序的启动运行。答案为 D。

2. 以下关于异常说法正确的是（ ）。

- A. 程序中抛出异常一定会引发程序中止
- B. 程序中抛出异常不一定会引发程序中止

- C. 拼写错误会导致程序中止
- D. 缩进错误会导致程序中止

【解析】程序中抛出异常只要被捕捉就不会中止运行，而拼写错误和缩进错误的程序根本就启动不了。答案为 B。

3. 关于以下 Python 程序段说法错误的是 ()。

```
try:
    #语句块 1
except IndexError:
    #语句块 2
```

- A. 该程序捕获了异常，因此一定不会因引发异常而中止
- B. 该程序捕获了异常，但不一定会因引发异常而中止
- C. 在#1 处语句中引发 IndexError 不会因引发异常而中止
- D. 在#2 处的语句不一定会被执行

【解析】虽然该程序段捕获了 IndexError，但是当引发其他异常时也会中止程序。答案为 A。

4. 关于以下 Python 程序段说明错误的是 ()。

```
try:
    #语句块 1
except:
    #语句块 2
else:
    #语句块 3
finally:
    #语句块 4
```

- A. 语句块 1 不一定会被执行
- B. 语句块 2 一定会被执行
- C. 语句块 3 一定会被执行
- D. 语句块 4 一定会被执行

【解析】在 try 语句中，finally 后的语句块可以用于保证释放资源或相关操作，因此一定会执行。答案为 D。

5. 以下关于 Python 程序调试与测试，说法错误的是 ()。

- A. pdb 模块具有设置断点功能调试程序的功能
- B. pdb 模块可以全面测试程序
- C. doctest 模块可以测试程序
- D. doctest 模块可以调试程序

【解析】pdb 模块用于程序调试。答案为 B。

二、实验题

1. 编程实现对用户输入的整数进行求和与求平均值（当用户输入的不是整数时应跳过，并进行相应的提示，而不能中止程序）。

2. 请你对第 5 章习题中的实验题第 1 题、第 3 题的程序进行调试和测试。

3. 请你对第 6 章习题中的实验题第 1 题、第 2 题的程序进行调试和测试。

第 2 篇 Python 编程高阶

第 8 章 复杂程序组织

当一个应用程序比较简单时，将程序代码写入一个文件即可。但随着应用程序或项目复杂度增加，如果将所有代码都写入同一个文件中，会出现文件过长或过大的情况，即不方便代码浏览，也不方便代码的管理、使用与维护。此时，很自然地会将同一个应用程序或项目内容按照功能或其他标准，分别放入不同的文件。不同的代码文件就是不同的模块，换句话说，每个“.py”文件都是一个模块。

以模块方式组织代码能够方便地管理和维护代码，如果项目的复杂度进一步增加，则模块可能也不能胜任了。于是，将项目中不同功能的代码放入不同的文件夹中，它们可以相互引用，这就是包。

模块和包都是复杂程序组织的一种方式，一般来说，复杂度较低的使用模块来管理即可，而复杂度较高的还要用到包来管理代码。

本章主要介绍通过 Python 中模块与包组织复杂的代码方法，内容包括：

- 模块概述；
- 模块的用法；
- 默认 Python 标准模块位置；
- 模块对编程的影响；
- 包概述；
- 常见应用程序组织方式。

8.1 模块

Python 中的模块实际上就是包含函数或者类的 Python 程序，一个大型的程序经常将功能细化，就是将实现不同功能的代码放在不同的程序中实现，在其他的程序中以模块的形式使用细化的功能，这样便于程序的维护和重用。



8.1.1 模块概述

模块是包含函数和其他语句的 Python 脚本文件，它以“.py”为后缀名，也就是 Python 程序的后缀名。用作模块的 Python 程序与其他的程序并没有什么区别。

使用模块中的代码，也很简单，那就是通过导入模块，然后使用模块中提供的函数或者数据。

在 Python 中可以使用以下三种方法导入模块或者模块中的函数：

```
import 模块名
import 模块名 as 新名字
from 模块名 import 函数名
```

其中使用 `import` 是将整个模块导入，而使用 `from` 则是将模块中某一个函数或者名字导入，而不是整个模块。使用 `import` 和 `from` 导入模块还有一个不同：使用 `import` 的导入模块，要使用模块中的函数则必须以模块名加“.”，然后是函数名的形式调用函数；而使用 `from` 导入模块

中的某个函数,则可以直接使用函数名调用,不用在前面加上模块名称。

此外,使用 `from` 导入时,函数名处可以只用一个“*”来表示导入该模块中所有代码。但要注意导入的模块中不要与此文件中的代码重复。

使用“`import 模块名 as 新名字`”用来在导入模块时给模块重新命一个名字,可能是为了防止名称重复,也可能是为了重新命一个简洁的名字,方便书写。

导入一个模块时,会创建新的命名空间,就可以使用命名空间来调用其中的代码;同时,还会在新创建的命名空间中执行模块中包含的代码,如果有输出也可以在控制台看到。

本书第 5.4 节的 `lambda` 实例中就使用了一次导入模块的功能。

【实例 8-1】演示了三种导入方式的使用方法,代码如下:

```
import math                                #直接导入math模块
from math import sqrt                     #只导入math模块的sqrt()函数,可直接使用
import math as shuxue                     #导入math模块并重命名为shuxue

print('调用math.sqrt:\t',math.sqrt(2))    #直接导入方式的调用
print('直接调用sqrt:\t',sqrt(2))          #调用仅导入的函数
print('调用shuxue.sqrt:\t',shuxue.sqrt(2)) #调用重命名
```

【代码说明】代码中使用了三种不同的方式导入 `math` 模块或其中的函数,然后分别调用了三种不同方式导入的对象。它们虽然都是导入同一个模块或模块中的内容,但并不冲突。

【运行效果】如图 8.1 所示,三次调用的输出都相同,因为实际上三个都是使用了 `math.sqrt()` 函数。

```
>>>
调用math.sqrt:      1.4142135623730951
直接调用sqrt:      1.4142135623730951
调用shuxue.sqrt:   1.4142135623730951
```

图 8.1 调用三种不同方式导入的对象

8.1.2 自己编写模块

自己编写模块其实和平常写 Python 程序是相同的,它既可以是一个解决某个问题的独立程序,也可以由几个函数构成。而模块的名称就是代码保存的文件名。

【实例 8-2】编写一个模块,导入并调用其中的函数,代码如下:

```
#模块文件
#文件名称: module_test.py
print('导入的测试模块的输出')          #导入时会被执行,输出信息

name = 'module_test'                    #定义一个变量
def m_t_pr():                             #模块中的函数定义
    print('模块module_test中m_t_pr()函数')

#调用自己编写的模块
#文件名: a8_2.py
import module_test                        #导入模块

module_test.m_t_pr()                     #调用导入模块的函数
print('使用module_test模块中的变量:',module_test.name) #使用导入模块中的变量
```



【代码说明】代码其实是保存在两个文件中的，在被使用为模块的文件代码中首先输出本模块信息，然后定义了一个变量和一个函数。导入模块后的代码只有两行，第一行调用导入模块中的函数，第二行直接输出导入模块中的变量。

【运行效果】如图 8.2 所示，第一行的输出实际上是导入 module_test 模块时，执行模块 module_test 的输出(module_test.py 代码中第一行)，第二行为调用 module_test 模块中的 m_t_pr() 函数的输出。

```
>>>
导入的测试模块的输出
模块module_test中m_t_pr()函数
使用module_test模块中的变量： module_test
```

图 8.2 调用自己编写的模块

由此可以看出：在模块中定义的变量同样可以在其他程序中导入并使用。

8.1.3 模块位置

编写好的模块只有被 Python 找到才能被导入。上一节中编写的模块和调用模块的程序位于同一个目录中，因此不需要进行设置就能被 Python 找到并导入。如果在该目录中新建一个 module 目录，并且把 module_test.py 转移到 module 目录中。再次在 Windows 的命令窗口中运行 a8_2.py，会引发 ImportError 错误，即找不到要导入的模块。

ImportError 错误表示：Python 解释器没有找到 module_test 模块。在导入模块时，Python 解释器首先在当前目录中查找要导入的模块。如果未找到模块，Python 解释器会从 sys 模块中的 path 变量指定的目录查找导入模块。如果在以上所有目录中未找到导入的模块，则会引发 ImportError 错误。

一般来说，Python 解释器在运行程序前将当前目录添加到 sys.path 路径列表中，所以导入模块时首先查找的路径是当前目录下的模块。在 Windows 系统下，其他的默认模块查找路径为 Python 的安装目录及几个子目录，如 lib、lib\site-packages、dlls 等。在 Linux 系统下，默认模块查找路径为/usr/lib、/usr/lib64 及它们的几个子目录。

模块的查找路径在 Python 中可以通过 sys 模块来进行操纵（查看、增加和删除），代码如下：

```
import sys                #导入 sys 模块

print(sys.path)           #输出当前模块查找路径（列表形式）
sys.path.append(Apath)    #添加 Apath 为模块查找路径
```

因此，要想在 Python 运行时能够查找到自定义的模块，可以通过在程序中操纵 sys.path 列表来实现。如上所述，若将 module_test.py 转移到 module 目录中，实例 8-2 代码应修改为以下代码：

```
#模块文件
#文件名称: module\module_test.py
print('导入的测试模块的输出')                                #被导入时会被执行，输出信息

name = 'module_test'                                         #定义一个变量
def m_t_pr():                                                #模块中的函数定义
    print('模块module_test中m_t_pr()函数')

#调用自己编写的模块
```

```
#文件名: a8_2.py
import sys
sys.path.append('D:\\lx\\module')
import module_test

module_test.m_t_pr()
print('使用 module_test 模块中的变量: ',module_test.name)
```

```
#导入 sys 模块
#添加模块查找路径
#导入 module_test 模块

#调用导入模块的函数
#使用导入模块中的变量
```



注意 添加模块查找路径时使用绝对路径。

8.1.4 __pycache__ 目录

在上一节的例子中,运行完 usemodule.py 会发现 moudle 目录中除了 mymodule.py 文件以外还多了一个目录 __pycache__, 目录下有一个 module_test.cpython-34.pyc 文件。其中 module_test.cpython-34.pyc 就是 Python 将 module_test.4.py 编译成字节码的文件。虽然 Python 是脚本语言,但 Python 可以将程序编译成字节码的形式。对于模块,Python 总是在第一次调用后将其编译成字节码的形式,以提高程序的启动速度。

Python 在导入模块时会查找模块的字节码文件,如果存在则将编译后的模块修改时间同模块的修改时间相比较。如果两者的修改时间不相符,那么 Python 将重新编译模块,以保证两者内容相符。被编译的程序也是可以直接运行的。

当然,没有必要去刻意编译 Python 程序。不过,由于 Python 是脚本,如果不想将源文件发布,可以发布编译后的程序,这样可以起到一定的保护源文件的作用。

对于不作为模块的程序,Python 不会在运行脚本后将其编译成字节码的形式。如果想将其编译,可以使用 compile 模块。以下代码可以将上一节中的 a8_2.pyc 编译成“.pyc”文件:

```
# file: compile.py
#
import py_compile;
py_compile.compile(' a8_2.py',' a8_2.pyc');
```

```
# 导入 py_compile 模块
# 编译 a8_2.py
```

运行 compile.py 后,可以看到当前目录中多了一个 a8_2.pyc 文件。在 Python3 中,如果在 py_compile.compile 函数中不指定第 2 个参数,则将在当前目录新建一个名为“__pycache__”的目录,并在这个目录中生成“被编译模块名.cpython-32.pyc”的 pyc 字节码文件。

运行 a8_2.pyc 后的输出和图 8.2 是相同的,编译后生成的 a8_2.pyc 并没有改变程序功能,只是以 Python 字节码的形式存在。

另外可以通过 Python 的命令行选项将脚本优化编译。Python 编译的优化选项有以下两个:

- -O 该选项对脚本的优化不多,编译后的脚本以“.pyo”为扩展名。凡是以“.pyo”为扩展名的 Python 字节码都是经过优化的;
- -OO 该选项对脚本优化的程度较大。使用该标志可以使得编译的 Python 脚本更小。使用该选项可以导致脚本运行错误,因此,应谨慎使用。

可以通过在命令行中输入以下命令将 a8_2.py 优化编译:

```
python -O compile.py
python -OO compile.py
```

8.1.5 具有独立运行能力的模块

每个 Python 程序在运行时都有一个 __name__ 属性 (name 前后均是两条下画线)。在程序中通过对 __name__ 属性值的判断,可让程序在作为导入模块和独立运行时都能正确运行。



在 Python 中，如果程序作为模块被导入，则其 `__name__` 属性被设置为模块名。如果程序独立运行，则其 `__name__` 属性被设置为 “`__main__`”。因此可以通过 `__name__` 属性来判断程序的运行状态。

如对实例 8-2 代码进行修改，它既可以独立运行，又可以作为模块被其他程序导入使用，修改后代码如下：

```
#模块文件
#文件名称: module\module_test.py
print('导入的测试模块的输出')                                #被导入时会被执行，输出信息

name = 'module_test'                                         #定义一个变量
def m_t_pr():                                                #模块中的函数定义
    print('模块module_test中m_t_pr()函数')

if __name__ == '__main__':
    m_t_pr()
    print(name)
```

一般来说，将模块的主要功能以实例的形式放在这个 if 语句中，可以方便测试模块是否能正常运行，或者发现模块的错误。当然这是一个好习惯，如果作为一个主程序，也可以不使用它。

此外，如果想了解模块中提供的功能（变量名、函数名），可使用内建的函数 `dir(模块名)` 来输出模块中的这些信息，当然也可以不使用模块名参数来列出当运行时中的模块信息。用 `dir()` 列出的 `module_test` 模块中的信息如图 8.3 所示。

```
>>> dir(module_test)
['_builtins_', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'm_t_pr', 'name']
```

图 8.3 用 `dir()` 显示模块信息

8.2 包

Python 中的模块实际上就是包含函数或者类的 Python 程序。对于一个大型的程序经常要将功能细化，把实现不同功能的代码放在不同的程序中实现，在其他的程序中以模块的形式使用细化的功能，这样便于程序的维护和重用。



8.2.1 包概述

当应用程序或项目具有较多的功能模块时，如果把它们都放在同一个文件夹下，就显得不合理了。这时，可以使用 Python 中提供的包来管理较多的功能模块。使用包的好处在于可以有效避免名字冲突，便于包的维护管理。

包其实就是一个文件夹或目录，但其中必须包含一个名为 “`__init__.py`”（`init` 的前后均是两条下画线）的文件。“`__init__.py`” 可以是一个空文件，仅用于表示该目录是一个包。此外，包还可以嵌套，即把子包放在某个包中。

包可以看作处于同一目录中的模块。在 Python 中首先使用目录名，然后使用模块名导入所需要的模块。要导入子包必须依照包顺序（目录顺序）以点分隔使用 `import` 进行导入。

例如，对于一个 web 项目，可能的包组织结构如下：

```
mywebproj/
```

```

manage.py          #主程序
urls.py
__init__.py
handle/
    __init__.py
    index.py
    info.py
template/
    index.html
    info.html
tools/
    __init__.py
    send_email.py
    graphic.py

```

如果在主程序中调用 `handle` 包中的 `index.py` 模块中的 `hdl()` 函数, 使用以下三种方法导入:

```

import handle.index          #导入后应使用 handle.index.hdl() 调用
from handle import index     #导入后应使用 index.hdl() 调用
from handle.index import hdl #导入后应使用 hdl() 调用

```

8.2.2 包详解

上一节中所述的包目录中所属的 `__init__.py` 文件是一个空文件, 只不过是作为包的一个标志。

实际上在第一次导入包中的任何部分, 就会执行 “`__init__.py`” 文件中的代码, 其中的变量和函数等也会自动导入。“`__init__.py`” 文件中也可以包含可执行包的初始化工作的代码和设置 “`__all__`” 变量。对于在 `from` 中使用 “`*`” 通配符导入包内所有名字时, 在 “`__init__.py`” 中设置 “`__all__`” 变量可以保证名字的正确导入。

【实例 8-3】 演示了一个包结构的程序, 代码如下:

```

#当前目录下 pcka 子目录中
#__init__.py 文件的内容
name = 'pcka'          #定义变量 name
print('__init__.py 中输出:', name) #输出变量 name 的值

def pck_test_fun():    #定义函数 pck_test_fun()
    print('pcka 包中的方法 pck_test_fun')

#以下是主程序的内容
#文件名: a8_3.py
import pcka            #导入包 pcka

print("输出 pcka 包中的变量 name:", pcka.name) #调用并输出包 pcka 中变量 name

print('调用 pcka 包中的函数: ', end='')
pcka.pck_test_fun()    #调用包 pcka 中的函数

```

【代码说明】 代码中显示, 标志包的文件 `__init__.py` 也有了代码, 包括定义变量 `name` 并输出, 定义了一个函数。而在主程序代码中, 只有一句 “`import pcka`”, 使得 `__init__.py` 中的代码被执行, 并自动导入了其中的变量和函数。

【运行效果】 如图 8.4 所示, 在导入后, `__init__.py` 得到运行并输出。随后直接调用了自动导入的变量和函数。



```
>>>
__init__.py中输出: pcka
输出pcka包中的变量name: pcka
调用pcka包中的函数: pcka包中的方法pck_test_fun
```

图 8.4 通过导入运行__init__.py 并自动导入其中的变量和函数

当使用了包之后，包中的模块也可能需要相互引用。以下面的包结构为例：

```
grnd\
  __init__.py
  prnta
    __init__.py
    suba.py
    sub\
      __init__.py
      sona.py
  prntb
    __init__.py
    subb.py
    subc.py
```

包中的模块要引用同一目录下的另一模块，可以直接导入。如 subb.py 要调用 subc.py 中的代码，使用以下语句：

```
import subc
```

如果一个模块要调用其同级别包中的另一个模块，就必须从父包上级开始导入。如 subb.py 要调用 suba.py 中的代码，使用以下语句：

```
from grnd.prnta import subc
```

如果一个模块要调用其目录下的子包中某个模块，可以使用相对导入的方式。如 suba.py 要调用 sona.py 中的代码，使用以下语句：

```
from .sub import sona
```

8.3 Python 常用标准库简介

Python 语言中内置的标准库包含 200 个左右的包和模块，提供了广泛的功能，涵盖数据处理、压缩和散列、数据持久化、数据库操作、文件及文件系统处理、日志系统、网络通信、Internet 协议、数据编码与转换、多进程与多线程等库和模块。除此以外，Python 还有大量的第三方库可以使用。本节简单介绍几个常用的标准库，还有一些标准库将会在本书的后续章节加以详细介绍。

8.3.1 数学类模块

此类常用的标准库有 math、random。

math 中有大量的常见数学计算函数，比如三角函数(sin()、cos()、tan())，反三角函数(asin()、acos()、atan())，对数函数(log()、log10()、log2())，还包括数学中的常量，如 e、pi（圆周率）。

random 中包含了常见的随机数生成函数，如 random、randint，还包括一些按概率生成随机数的函数如 gauss 等，此外还有像 shuffle（乱序列表）、choice()（从序列中随机取元素）等随机函数。

在交互式环境下使用 random 模块示例代码如下：

```
>>> random.random()           #生成 0—1 之间随机数
0.5060695163308423
```

```

>>> random.randint(0,10)           #生成 0—10 之间随机整数
2
>>> random.randint(0,10)
1
>>> random.choice((1,2,3,4))       #从元组中随机返回一个值
2
>>> random.choice((1,2,3,4))
3
>>> alst = [1,2,3,4,5,6]
>>> random.shuffle(alst)            #对列表 alst 随机乱序（该函数返回值为 None）
>>> alst
[3, 2, 4, 6, 1, 5]                 #乱序后的列表

```

8.3.2 日期与时间类

此类标准库有 `calendar`、`datetime` 和 `time` 三个模块。

常需要获取的时间有两种：

```

time.time()           #获取自初始时间至现在的秒数
datetime.datetime.now() #获取本地的日期/时间
datetime.datetime.utcnow() #获取当前的 UTC 日期/时间

```

此外，还可以计算时间差或者给时间加上天数、小时等来计算未来的时间表示，在交互式环境下的示例代码及运行效果如下：

```

>>> import datetime           #导入模块 datetime
>>> datetime.datetime.now()    #调用函数，获取当前时间
datetime.datetime(2015, 3, 7, 23, 14, 56, 139627)
>>> import time               #导入模块 time
>>> time.time()
1425741308.728849
>>> t1 = datetime.datetime(2015, 3, 7, 23, 14, 56, 139627) #自定义时间
>>> t1
datetime.datetime(2015, 3, 7, 23, 14, 56, 139627)
>>> t2 = datetime.datetime.now()
>>> t2
datetime.datetime(2015, 3, 7, 23, 24, 2, 608587)
>>> sub = t2 - t1              #时间相减
>>> sub.seconds                #时间相差秒数
546
>>> t2.year,t2.month,t2.day,t2.hour,t2.minute,t2.second #时间中的年份、月份等
(2015, 3, 7, 23, 24, 2)

```

8.4 小结

本章主要介绍了 Python 语言中组织复杂程序或代码的基本方法。对于功能单元不多，而且功能单元中的代码少的小型应用，使用模块就可以管理好代码了。而对于功能单元多的中型应用程序或项目，应该用包和模块来组织代码。通过学习本章内容，你应掌握用包和模块来组织代码的方法，并了解如何探查包或模块的结构。

8.5 本章习题

一、选择题

1. 以下导入模块方式中错误的是（ ）。

A. `import mo`

B. `from mo import *`



- C. `import mo as moo` D. `import * from mo`

【解析】应掌握 Python 语言中导入模块和包的基本语法。答案为 D。

2. 对于以下导入方式, 使用模块 `math` 中的 `sqrt` 函数的正确方式为 ()。

`import math as mymath`

- A. `math.sqrt(3)` B. `sqrt(3)`
C. `mymath.sqrt(3)` D. `math.mymath.sqrt(3)`

【解析】使用该方式导入时, 模块的名字被重新命名而不能使用原先的名称。答案为 C。

3. 以下关于模块说法错误的是 ()。

- A. 模块就是一个普通的 Python 程序文件
B. 任何一个普通的 Python 程序文件都可以作为模块导入
C. 模块文件的扩展名可以是 .txt
D. Python 运行时只会从指定的目录搜索导入的模块

【解析】作为模块文件的扩展名一定要使用 .py。答案为 C。

4. 以下关于包说法错误的是 ()。

- A. 包可以是一个任何目录
B. 包是可以嵌套的
C. 作为包中的目录要包含特殊的 `__init__.py` 文件
D. 包目录中的 `__init__.py` 文件内容可以为空

【解析】包目录中必须含有 `__init__.py` 文件。答案为 A。

5. 以下关于包和模块说法错误的是 ()。

- A. 包中可以包含模块
B. 模块中可以包含包
C. 一般在小规模的项目中使用模块就行了
D. 包一般需要在大规模的项目中使用

【解析】模块为一个程序文件, 而包是一个含 `__init__.py` 文件的目录。因此, 包中可以包含模块。答案是 A。

二、实验题

1. 编写一个模块来包含数学中的求平方、求绝对值、判断素数, 并且这个模块可以独立运行并输出一个数的平方、绝对值及是否素数。

【提示】独立运行的模块, 需要将运行的程序段放入 `if __name__ == "__main__":` 中。

2. 如果某个主项目需要两个子项目, 一个是第 1 题中的数学运算, 另一个是定制输出 (格式如下), 请使用包来组织它们, 并在主项目调用它们进行测试。

+*****+

计算结果:

+*****+

3. 定义一个类 `Person`, 具有实例属性 (姓名、生日、身高、体重) 和实例方法 (分别输出这个人的年龄、体重是否达标)。

第 9 章 迭代器、生成器与装饰器

迭代器、生成器与装饰器是 Python 语言中常用的语法形式。

迭代器的使用简化了循环程序的代码并可以节约内存，生成器的使用也可以节约大量的内存，特别是需要生成大量序列的对象时。迭代器是一种可以从其中连续迭代的容器，如前文所述，所有的序列类型都是可迭代的。而生成器则是函数中包含 `yield` 语句的一类特殊的函数。

装饰器的灵活性很强，可以为一个对象添加新的功能或者给函数插入相关的功能。在具体的程序设计中，可以灵活地设计出所需要的功能。

本章主要介绍 Python 中的迭代器、生成器和装饰器，内容包括：

- 迭代器及其创建；
- 生成器创建及其应用；
- 装饰器概念；
- 应用函数装饰器；
- 应用类装饰器。

9.1 迭代器



在 Python 中，迭代器的使用是最为广泛的，在本章以前的代码中，凡是使用 `for` 语句，其本质上都是迭代器的应用，本节主要介绍迭代器的概念、创建与应用。

9.1.1 迭代器概述

迭代器从表面上看是一个数据流对象或容器，当使用其中的数据时，每次从数据流中取一个数据，直到数据被取完，而且数据不会被重复使用。

从代码的角度看，迭代器是实现了迭代器协议方法的对象或类。迭代器协议方法主要有两个：

- `__iter__()`；
- `__next__()`。

“`__iter__()`”方法返回对象本身，它是 `for` 语句使用迭代器的要求。

“`__next__()`”方法用于返回容器中下一个元素或数据。当容器中的数据用尽时，应该引发 `StopIteration` 异常。

任何一个类，只要它实现了或具有这两个方法，就可以称其为迭代器，也可以说是可迭代的。在使用它作为迭代器时，就可以用 `for` 来遍历（迭代）它，示例代码如下：

```
for item in iterator:
    pass
```

在每个循环中，`for` 语句都会从迭代器的序列中取一个数据，并赋给 `item`，供循环体内使用或处理。从形式上看完全与遍历元组、列表、字符串、字典等序列一样。



9.1.2 自定义迭代器

如上一节所述，要定义一个自己的迭代器，只要定义一个实现迭代器协议方法的类即可。

【实例 9-1】自定义一个迭代器的演示实例，其代码如下：

```
class MyIterator:                                #自定义类迭代器类 MyIterator

    def __init__(self,x=2,xmax=100):             #定义构造方法，初始化实例属性
        self.__mul,self.__x = x,x
        self.__xmax = xmax

    def __iter__(self):                           #定义迭代器协议方法，返回类自身
        return self

    def __next__(self):                           #定义迭代器协议方法
        if self.__x and self.__x != 1:
            self.__mul *= self.__x
            if self.__mul <= self.__xmax:
                return self.__mul                #返回值
            else:
                raise StopIteration              #引 StopIteration 错误
        else:
            raise StopIteration

if __name__ == '__main__':
    myiter = MyIterator()                        #实例化迭代器 MyIterator
    for i in myiter:                             #遍历并输出值
        print('迭代的数据元素为: ',i)
```

【代码说明】代码中首先定义了迭代器类 `MyIterator`，在其构造方法中，初始化两个私有的实例属性，用于产生序列和控制序列产生的最大值。该迭代器总是返回所给数的 n 次方，但其最大值超过 `xmax` 参数值的大小时，就会引发 `StopIteration` 错误，并结束遍历。最后，实例化迭代器类，并遍历迭代器的值序列，同时进行输出。

从代码中可以看出：如果需要产生很大范围的数的序列，采用列表或元组等进行一次性生成，则会占据大量的内存空间，对系统有很大的“杀伤力”。而使用了迭代器，则可以每次调用时生成一个，显然可以节约大量的内存空间。



注意 迭代器类一定要在某个条件下引发 `StopIteration` 错误，以结束遍历循环，否则会产生死循环。

【运行效果】如图 9.1 所示，初始化迭代器时使用了默认参数，遍历得到的序列是 2 的 n 次方的值，最大值不超过 100。

```
>>>
迭代的数据元素为: 4
迭代的数据元素为: 8
迭代的数据元素为: 16
迭代的数据元素为: 32
迭代的数据元素为: 64
```

图 9.1 自定义迭代器输出信息

9.1.3 内置迭代器工具

在 Python 语言中，已经内建了一个用于产生迭代器的函数 `iter()`，另外在标准库的 `itertools`

模块中还有丰富的迭代器工具，它们存在于 `itertools` 模块中。

1. 内建迭代器函数 `iter()`

内建的 `iter()` 函数具有两种使用方式，其原型如下：

```
iter(iterable)
iter(callable, sentinel)
```

第一种原型中只有一个参数，要求参数为可迭代的类型，当然也可以使用前文所说的各种序列类型。

第二种原型中具有两个参数，第一个参数是可调类型，一般为函数；第二个参数被称为“哨兵”，即当第一个参数（函数）调用返回值等于第二个参数的值时，迭代或遍历停止。

【实例 9-2】 演示了第二种原型的使用方法，代码如下：

```
class Counter:                                #定义用于计数的类
    def __init__(self,x=0):                    #定义构造函数，初始化实例属性 x
        self.x = x

counter = Counter()                            #实例化类 Counter

def used_iter():                               #定义用于 iter() 函数的函数
    counter.x += 2                             #修改计数类中的实例属性的值
    return counter.x

for i in iter(used_iter,8):                    #迭代 iter() 函数产生的迭代器
    print('本次遍历的数值: ',i)
```

【代码说明】 代码中首先定义一个用于计数的类 `Counter`，用于记录当前值，并实例化这个类作为全局变量；然后定义一个在 `iter()` 函数中调用的函数，用 `for` 来遍历 `iter()` 产生的迭代器并输出遍历得到的值。

【运行效果】 如图 9.2 所示，分别遍历得到 2、4、6，当再次计算时得到 8，与 `iter()` 函数中提供的第二个参数 8 相等，此时，停止了迭代。

```
>>>
本次遍历的数值: 2
本次遍历的数值: 4
本次遍历的数值: 6
```

图 9.2 遍历 `iter()` 迭代器

2. `itertools` 模块中常用工具函数

`itertools` 模块中提供了近二十个迭代器工具函数，主要有三类。

无限迭代器：

```
count(start,[step])                #从 start 开始，以 step 为步进行计数迭代
cycle(seq)                          #无限循环迭代 seq
repeat(elem,[n])                   #循环迭代 elem
```

迭代短序列：

```
chain(p,q,...)                     #链接迭代（将 p 和 q 连接起来迭代，就像从一个序列中迭代）
compress(data,selectors)            #依据 selectors 中的值选择迭代 data 序列中的值
dropwhile(pred,seq)                 #当 pred 对序列元素处理结果为假时开始迭代 seq 后所有值
filterfalse(pred,seq)               #当 pred 处理为假的元素
```



```

takewhile(pred,seq)          #与dropwhile相反
tee(it,n)                    #将it重复n次进行迭代
zip_longest(p,q,...)

```

组合迭代序列

```

product(p,q,...[,n])         #迭代排列出所有的排列
permutations(p,r)            #迭代序列中r个元素的排列
combinations(p,r)           #迭代序列中r个元素的组合

```

以下代码在交互式环境下演示了 `itertools` 模块中的常用迭代工具函数的使用方法及输出信息:

```

>>> import itertools
>>> for i in itertools.count(1,3):
    print(i)
    if i >= 10:
        break

1
4
7
10
>>> x = 0
>>> for i in itertools.cycle(['a','b']):
    print(i)
    x += 1
    if x >= 6:
        break

a
b
a
b
a
b
>>> list(itertools.repeat(3,3))
[3, 3, 3]
>>> list(itertools.chain([1,3],[2,3]))
[1, 3, 2, 3]
>>> list(itertools.compress([1,2,3,4],[1,[],True,3]))
[1, 3, 4]
>>> list(itertools.dropwhile(lambda x:x>6,[8,9,1,2,8,9]))
[1, 2, 8, 9]
list(itertools.takewhile(lambda x:x>10,[18,19,1,21,8,9]))
[18, 19]
>>> for its in itertools.tee([0,1],2):
    for it in its:
        print(it)

0
1
0
1
>>> list(itertools.permutations('abc',2))
[('a', 'b'), ('a', 'c'), ('b', 'a'), ('b', 'c'), ('c', 'a'), ('c', 'b')]
>>> list(itertools.combinations('abc',2))
[('a', 'b'), ('a', 'c'), ('b', 'c')]

```



注意

使用无限迭代器时, 必须有迭代退出的条件, 否则会导致死循环。

9.2 生成器



使用生成器，可以生成一个值的序列用于迭代，并且这个值的序列不是一次生成的，而是使用一个，再生成一个，的确可以使程序节约大量内存。

9.2.1 生成器创建

生成器对象是使用 `yield` 关键字定义的函数对象，因此，生成器也是一个函数。生成器用于生成一个值的序列，以便在迭代中使用。

【实例 9-3】 演示了自定义生成器函数及其使用的实例，其代码如下：

```
def myYield(n):                                #定义一个生成器（函数）

    while n>0:
        print("开始生成...:")
        yield n                                #yield 语句，用于返回给调用者其后表达式的值
        print("完成一次...:")
        n -= 1

if __name__ == '__main__':
    for i in myYield(4):                        #for 语句遍历生成器
        print("遍历得到的值: ",i)
    print()
    my_yield = myYield(3)                       #生成一个生成对象
    print('已经实例化生成器对象')
    my_yield.__next__()                         #手工调用其特殊方法，获取序列中一个值
    print('第二次调用__next__()方法: ')
    my_yield.__next__()
```

【代码说明】 代码自定义了一个递减数字序列的生成器，每次调用时都会产生一个从调用时所提供值为初始值的不断递减的数字序列。生成对象可以直接被 `for` 遍历，也可以手工进行遍历，手工的方法见上面代码的最后两行。

【运行效果】 如图 9.3 所示，第一次使用 `for` 语句直接遍历自己创建的生成器，第二次用手工的方法获取生成器产生的数值序列。

```
>>>
开始生成...:
遍历得到的值: 4
完成一次...:
开始生成...:
遍历得到的值: 3
完成一次...:
开始生成...:
遍历得到的值: 2
完成一次...:
开始生成...:
遍历得到的值: 1
完成一次...:

已经实例化生成器对象
开始生成...:
第二次调用__next__()方法:
完成一次...:
开始生成...:
```

图 9.3 创建和使用生成器的运行结果



9.2.2 深入生成器

如上节所述，生成器中包含 `yield` 语句，可以用 `for` 直接遍历；也可以手工调用其 `__next__()` 方法进行遍历。

`yield` 语句是生成器中的关键语句，生成器在实例化时并不会立即执行，而是等待调用其 `__next__()` 方法才开始运行。并且当程序运行完 `yield` 语句后就会“吼（hold）住”，即保持其当前状态且停止运行，等待下一次遍历时才恢复运行。

如图 9.3 所示，在程序运行结果中的空行之后、输出“已经实例化生成器对象”之前，已经实例化了生成器对象，但生成器并没有运行（没有输出“开始生成”）。当第一次手工调用 `__next__()` 方法之后，才输出“开始生成”，标志着生成器已经运行，而在输出“第二次调用 `__next__()` 方法：”之前并没有输出“完成一次”，说明 `yield` 语句运行之后就立即停止了。而第二次调用 `__next__()` 方法之后，才输出“完成一次...”，说明生成器的恢复运行是从 `yield` 语句之后开始运行的。

`yield` 语句不仅可以使函数成为生成器和返回值，还可以接受调用者传来的数值。但值得注意的是：第一次调用生成器时不能传送给生成器 `None` 以外的值，否则会引发错误。

【实例 9-4】 演示了一个可以接收调用者传来的值并重新初始化生成器生成的值，代码如下：

```
def myYield(n):                                #定义一个生成器函数
    while n>0:
        rcv = yield n                          #rcv 用来接收调用者传来的值
        n -= 1
        if rcv is not None:                    #rcv 不为 None 就和重置 n 值为 rcv
            n = rcv

if __name__ == '__main__':
    my_yield = myYield(3)                      #初始化生成器
    print(my_yield.__next__())
    print(my_yield.__next__())
    print('传给生成器一个值，重新初始化生成器。')
    print(my_yield.send(10))
    print(my_yield.__next__())
```

【代码说明】 代码中首先定义了一个生成器函数，其中 `yield` 语句为“`rcv = yield n`”，`rcv` 就可以接收调用者传来的值。如果调用时提供了一个值，就会从这个值开始递减产生序列。

【运行效果】 如图 9.4 所示，开始遍历时从默认的 3 开始递减并输出；重新传一个值（10）给生成器时，得到的遍历从 10 开始递减。

```
>>>
3
2
传给生成器一个值，重新初始化生成器。
10
9
```

图 9.4 接收值的生成器运行结果

此外，除了内建的 `iter()` 与 `itertools` 模块，还有一种类似列表推导的生成器表达式，如图 9.5 所示，在交互式环境下第一行代码就返回一个生成器。

```
>>> a= (i for i in range(5))
>>> a
<generator object <genexpr> at 0x0000000036258B8>
>>> type(a)
<class 'generator'>
>>> list(a)
[0, 1, 2, 3, 4]
```

图 9.5 生成器表达式

生成器对象也可以通过 `list()` 函数转换为列表，如图 9.5 所示的最后一行指令就把 `a` 这个生成器对象转换为列表。应该注意的是，如果生成器生成序列是无限的或大量的序列，不可以将其转换为列表，否则会导致死机或大量消耗内存。

9.2.3 生成器与协程

上节所述的运用 `send()` 方法来重置生成器的生成序列，其实也被称为协程。协程是一种解决程序并发的方法。

如果采用一般的方法来实现生产者与消费者这个传统的并发与同步程序设计问题，则要考虑的问题还是比较繁杂的。但通过生成器实现的协程，解决这个问题就很简单。

【实例 9-5】 演示了一个简单的生产者与消费者编程模型，代码如下：

```
def consumer():
    # 定义一个消费者模型（生成器协程）
    print('等待接收处理任务...')
    while True:
        data = (yield)
        print('收到任务: ',data)
        # 模拟接收并处理任务
        # （此处可以执行函数调用来完成相关任务）

def producer():
    # 定义一个生产者模型
    c = consumer()
    c.__next__()
    for i in range(3):
        print('发送一个任务...', '任务%d' % i)
        c.send('任务%d' % i)

if __name__ == '__main__':
    producer()
```

【代码说明】 代码中定义的两个函数分别代表生产者和消费者模型，而其中的消费者模型实际是一个生成器。在生产者模型函数中每个循环模拟发送一个任务给消费者模型（生成器），而生成器则可以调用相关函数来处理任务。这一任务就是运用 `yield` 语句的“hold”特性来完成的。

【运行效果】 如图 9.6 所示，每次发送一个任务都是通过调用生成器的 `send()` 函数实现的，而收到任务的生成器会执行相关的函数调用并完成子任务。

```
>>>
等待接收处理任务...
发送一个任务... 任务0
收到任务: 任务0
发送一个任务... 任务1
收到任务: 任务1
发送一个任务... 任务2
收到任务: 任务2
```

图 9.6 协程示例输出



9.3 装饰器



装饰器是一种增加函数或类功能的简单方法，它可以快速地给不同的函数或类插入相同的功能。从本质上说，它是一种代码实现方式。

9.3.1 装饰器概述

为了给不同的函数或类插入相同的功能，在 Python 中可以使用优雅而简单的工具——装饰器。与其他高级语言相比，它简化了代码，并且可以快速实现所需要的功能。同时，它为函数或类对象增加功能也是透明的，对于同一函数，既可以添加简单的功能，也可以添加复杂的功能，运用起来很灵活。而在调用被装饰的函数时，没有任何附加的东西，仍然像调用原函数或没有被装饰的函数一样。

装饰器的表示语法是使用一个特殊的符号“@”来实现的。装饰器装饰函数或类就是用“@装饰器名称”放在函数或类的定义行之前即可。例如，有一个装饰器名称为 `disp_run_time`，在使用装饰器的函数定义时使用以下形式：

```
@disp_run_time
def decorated_fun():
    pass
```

使用了装饰器后，此处定义的函数 `decorated_fun()` 就可以只定义自己所需的功能，而装饰器所定义的功能会自动插入到函数 `decorated_fun()` 之中。这样，可以节约大量具有相同功能的函数或类的代码。即使不同目的、不同类的函数或类也可以插入相同的功能。

要用装饰器来装饰对象，必须先定义装饰器，装饰器的定义与普通函数的定义在形式上完全一致，只不过装饰器函数的参数必须有函数或类对象，然后在装饰器函数中重新定义一个新的函数或类，并在其中执行某些功能前后或中间来使用被装饰的函数或类，最后返回这个新定义的函数或类。以下就是一个简单的装饰器函数的定义代码：

```
def demo_decorator(fun):           #定义装饰器函数（参数为 fun，可接受函数对象）
    def new_fun(*args,**kwargs):   #新定义一个包装器函数用于返回
        pass                       #包装器函数中调用被装饰的函数
        fun(*args,**kwargs)
        pass
    return new_fun                 #返回包装器函数
```

此外，装饰器还可以嵌套装饰，比如以下代码中，函数 `decorated_fun()` 同时被 `abc` 和 `disp_run_time` 装饰器装饰，插入了两种不同类型的功能：

```
@abc
@disp_run_time
def decorated_fun():
    pass
```

9.3.2 装饰函数

用装饰器装饰函数，首先要定义装饰器，然后用定义的装饰器来装饰函数。

【实例 9-6】 演示了自定义一个装饰器并用来装饰自定义的函数，代码如下：

```
def abc(fun):                      #定义一个装饰器 abc
    def wrapper(*args,**kwargs):  #定义包装器函数
        print('开始运行...')
        fun(*args,**kwargs)      #调用被装饰函数
        print('运行结束!')
```

```

    return wrapper                                #返回包装器函数

@abc                                              #装饰函数语句
def demo_decoration(x):                          #定义普通函数，被装饰器装饰
    a = []
    for i in range(x):
        a.append(i)
    print(a)

@abc
def hello(name):                                #定义普通函数（被装饰器装饰）
    print('Hello ',name)

if __name__ == '__main__':
    demo_decoration(5)                          #调用被装饰器装饰的函数
    print()
    hello('John')                              #调用被装饰器装饰的函数

```

【代码说明】代码中首先定义了一个装饰器函数（abc），它带有一个可以使用函数对象的参数。接着定义了两个被装饰器装饰的普通函数（demo_decoration、hello）。最后对被装饰的函数进行调用。由此可以看出，调用被装饰的函数与调用普通函数没有任何区别。而在装饰器定义的内部，很明显又定义了一个内嵌的函数 wrapper()，在这个内嵌的函数中执行了其他的一些语句，也调用了被装饰的函数。最后返回了这个内嵌的函数，代替了被装饰的函数，从而完成了装饰器的功能。

【运行效果】如图 9.7 所示，达到了在调用两个被装饰的函数前后输出了相应的信息的功能。

```

>>>
开始运行...
[0, 1, 2, 3, 4]
运行结束！

开始运行...
Hello John
运行结束！

```

图 9.7 装饰器定义与装饰函数的调用

装饰器也是可以带参数的，比如“@abc("callcall")”。可以将实例 9-6 中的装饰器函数改写为带参数的装饰器，代码如下：

```

def abc(action):
    def mabc(fun):
        def wrapper(*args,**kwargs):
            print('开始运行...',action)
            fun(*args,**kwargs)
            print('运行结束!',action)
        return wrapper
    return mabc

```

对比实例 9-6 中的装饰器函数可以看出，在装饰器函数中嵌套了两层的函数，一层层向外返回函数。最外层的参数并不是可调用类型的参数。

9.3.3 装饰类

装饰器不仅可以装饰函数，也可以装饰类。定义装饰类的装饰器，采用的方法是：定义内嵌类的函数，并返回新类。



【实例 9-7】演示了一个类装饰器及其使用的例子，代码如下：

```
def abc(myclass):                                #定义类装饰器
    class InnerClass:                            #定义内嵌类
        def __init__(self,z=0):
            self.z = 0
            self.wrapper = myclass()            #实例化被装饰的类

        def position(self):
            self.wrapper.position()
            print('z axis:',self.z)

    return InnerClass                            #返回新定义的种类

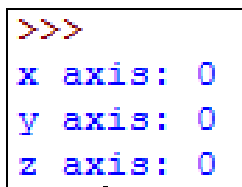
@abc                                             #应用装饰器
class coordination:                             #定义普通的类
    def __init__(self,x=0,y=0):
        self.x = x
        self.y = y

    def position(self):
        print('x axis:',self.x)
        print('y axis:',self.y)

if __name__ == '__main__':
    coor = coordination()                      #实例化被装饰的类
    coor.position()                           #调用方法 position()
```

【代码说明】代码中首先定义了一个能够装饰类的装饰器 `abc`；在其中定义了一个内嵌类 `InnerClass` 用于代替被装饰的类，并返回新的内嵌类。而在实例化普通类时，得到的就是被装饰器装饰后的类。

【运行效果】如图 9.8 所示，原来定义的坐标类只包含平面坐标，而通过装饰器的装饰后，则成为了可以表示立体坐标的三个坐标值。在图中可以看出显示的坐标为立体坐标值。



```
>>>
x axis: 0
y axis: 0
z axis: 0
```

图 9.8 装饰器装饰类的输出

9.4 小结

本章主要介绍了 Python 语言中比较常用的语法，即迭代器、生成器和装饰器。首先介绍了定义迭代器应实现其协议方法 `__iter__()` 和 `__next__()`，并且应注意当需要结束迭代时抛出 `StopIteration` 错误即可，之后介绍了 Python 的内置迭代器工具 `iter` 和 `itertools` 模块。其次介绍了生成器的创建方法、特点与协议方面的应用。最后介绍了装饰器的基本概念，以及如何用它来装饰函数和类，以实现某些功能。通过学习本章内容，你应掌握迭代器、生成器的特性及应用场合，会使用装饰器来为函数和类插入所需要的功能。

9.5 本章习题

一、选择题

1. 以下关于迭代器说法错误的是 ()。

- A. 迭代器数据用尽时抛出 `IndexError` 异常
- B. `__next__()` 是迭代器的协议方法
- C. `__iter__()` 是迭代器的协议方法
- D. 迭代器可以节约内存

【解析】迭代器数据用尽时应抛出 `StopIteration` 异常。答案为 A。

2. `for` 语句循环可以结束的条件不包括 ()。

- A. 使用完迭代的序列
- B. 循环体中运行了 `break` 语句
- C. 循环体中抛出了 `EOFError` 异常且未捕获
- D. 使用的迭代器中抛出 `StopIteration` 异常

【解析】循环体中抛出了 `EOFError` 异常且捕获则中止。答案为 C。

3. 以下关于生成器说法错误的是 ()。

- A. 生成器对象中应包含 `yield` 语句
- B. 生成器对象可以用 `for` 语句迭代
- C. 可以手工调用生成器对象的 `__next__` 方法
- D. 生成器对象在使用时占用内存和同序列的列表一样

【解析】使用生成器对象的目的就是减少内存的使用。答案为 D。

4. 以下关于装饰器说法错误的是 ()。

- A. 装饰器可以为不同的功能函数插入相同的功能
- B. 装饰器可以装饰函数
- C. 类装饰也可以装饰函数
- D. 装饰器装饰具体类或函数时，其顺序不同，结果也不同

【解析】类装饰器和函数装饰器不可以通用。答案为 C。

二、实验题

1. 自定义一个实现迭代生成小于某个整数的所有素数的迭代器。
2. 为了对函数的运行开始和结束分别给出一个提示，请定义一个装饰器，在函数运行开始时，输出“start...”；结束时，输出“end.”。
3. 使用生成器实现 `range` 函数的功能。

第 10 章 Python 进阶话题

在 Python 语言中，因其语言特性而产生了一些很有用的编程模式与编程方法。作为“Pythoner”，想要编写出 Pythonic 风格的程序，就要了解这些特别的编程方法，主要包括函数与其命名空间、闭包及其应用、上下文管理器、鸭子类型等。

本章主要介绍 Python 语言中的这些特性，内容包括：

- 函数与命名空间;
- 闭包概念;
- 闭包实现延迟求值;
- 闭包实现泛型函数;
- 上下文管理器;
- 用字符串操作对象属性;
- 鸭子类型与多态。

10.1 函数与命名空间



在 Python 中可以通过模块来管理复杂的程序，而将不同功能的函数分布在不同的模块中，那么函数及其全局命名空间决定了函数中引用全局变量的值。函数的全局命名空间始终是定义该函数的模块，而不是调用该函数的命名空间。因此，在函数中引用的全局变量始终是定义该函数模块中的全局变量。

【实例 10-1】演示了函数与其全局命名空间关系的实例，代码如下：

#foo 模块文件： foo.py	
name = "Foo module"	#定义全局变量 name
def foo_fun(): print('函数 foo_fun:') print('变量 name:',name)	#定义函数 foo_fun() #输出全局变量 name
#a10_1.py 文件 #导入 foo 模块函数	
from foo import foo_fun	#导入 foo 模块中的函数 foo_fun
name = 'Current module'	#定义全局变量 name
def bar(): print('当前模块中函数 bar:') print('变量 name:',name)	#定义普通函数 bar() #输出全局变量 name
def call_foo_fun(fun): fun()	#定义调用传入函数作为参数的函数 #调用传入的函数
if __name__ == '__main__': bar() print()	#调用本模块中定义的函数 bar()

```
foo_fun()                #调用从模块 foo 中导入的函数 foo_fun()
print()
call_foo_fun(foo_fun)
```

【代码说明】在 `foo` 模块的代码中定义了全局变量 `name` 和函数 `foo_fun()`，并在函数 `foo_fun()` 中输出全局变量 `name` 的值。在 `a10_1.py` 文件中，也定义了全局变量 `name` 和函数 `bar()`，并在函数 `bar()` 中输出全局变量 `name` 的值。之后，分别调用本模块中定义的函数 `bar()` 和从 `foo` 模块中导入的函数 `foo_fun()`，最后还定义一个把函数作为参数传入并调用的函数 `call_foo_fun()`。因为函数中引用的全局变量始终是定义该函数模块中的全局变量，所以第一次调用输出了当前模块中的全局变量 `name` 的值，而第二次调用从 `foo` 模块中导入的函数 `foo_fun()` 输出的则是在 `foo` 模块中的全局变量 `name` 的值，第三次调用 `call_foo_fun()` 函数，并把从 `foo` 模块中导入的函数 `foo_fun()` 作为参数传入其中进行调用，即使是在函数内部被调用，它仍然输出函数 `foo_fun()` 模块中全局变量 `name` 的值。

【运行效果】如图 10.1 所示，第一次输出的是当前模块中的全局变量 `name` 的值“Current module”，而第二次输出的是 `foo` 模块中的全局变量 `name` 的值“Foo module”，第三次输出的仍然是 `foo` 模块中的全局变量 `name` 的值“Foo module”。

```
>>>
当前模块中函数bar:
变量name: Current module

函数foo_fun:
变量name: Foo module
```

图 10.1 在函数中引用全局变量实例的运行结果



要注意分清函数全局命名空间（定义的模块）与引用的模块之间的关系。

10.2 闭包及其应用

闭包是 Python 语言中一种特殊的语法现象，使用闭包可以灵活地实现用其他语法不太容易实现的一些功能，比如实现延迟求值和定义泛型函数功能的实现。

10.2.1 闭包概述

闭包是指 Python 语言中将组成函数的语句和这些语句的执行环境打包到一起所得到的对象。当使用嵌套函数（函数中定义函数）时，闭包将捕获内部函数执行所需的整个环境。此外，嵌套的函数可以使用被嵌套函数中的任何变量，就像普通函数可以引用全局变量一样，而不需要通过参数引入。

【实例 10-2】演示了一个嵌套函数的例子，代码如下：

```
x = 14                #全局变量 x

def foo():            #定义函数（嵌套的外层函数）
    x = 3
    def bar():        #定义函数（嵌套的内层函数）
        print('x is %d' % x)    #引用变量 x（实际引用的是嵌套的外层函数中的 x）
    bar()              #调用嵌套的内层函数

if __name__ == '__main__':
    foo()
```




【代码说明】代码中定义了一个全局变量 `x`，在嵌套函数的外层函数 `foo()` 中，也定义了一个变量 `x`；在嵌套的内层函数 `bar()` 中引用的 `x` 变量应该是 `foo()` 中定义的 `x`。

【运行效果】如图 10.2 所示，嵌套函数可以直接引用其外层的函数中定义的变量 `x` 的值并输出，所以输出的值为 3，而不是全局变量 `x` 的值 14。

```
>>>
x is 3
```

图 10.2 嵌套函数的运行结果

另外，本书第 9 章所介绍的装饰器也是通过闭包来实现的。

10.2.2 闭包与延迟求值

闭包可以实现先将参数传递给一个函数，而并不立即运行，以达到延迟求值的目的。

【实例 10-3】演示了一个可以延迟求值的实例，其代码如下：

```
def delay_fun(x,y):                #定义一个可以延迟求值的函数
    def caculator():              #内部嵌套的函数
        return x + y              #直接返回要求的值
    return caculator               #返回内部嵌套的函数对象

if __name__ == '__main__':
    print('返回一个求和的函数，并不求和。')
    msum = delay_fun(3,4)          #调用外层函数，并不计算，实际返回一个函数对象
    print()
    print('调用并求和：')
    print(msum())                  #实际求值的调用
```

【代码说明】代码中采用嵌套函数来实现延迟求值的功能。在外层函数中，`delay_fun(x,y)` 返回嵌套函数对象 `caculator()`，把参数 `x`、`y` 传入外层函数中，这个嵌套的函数对象 `caculator()` 直接引用外层函数中的值，而不需要用参数形式传递。因此，只有第二次调用返回的函数时，才真正地执行计算，并返回计算结果。

【运行效果】如图 10.3 所示，第一次调用函数 `delay_fun(3,4)` 时并没有计算结果，而在第二次调用其返回的函数时，才得到计算结果。

```
>>>
返回一个求和的函数，并不求和。

调用并求和：
7
```

图 10.3 延迟求值演示程序的运行结果



注意 所谓延迟求值，就是函数返回的是一个函数，而真正需要运行函数中的代码时，其本质还是函数调用。

10.2.3 闭包与泛型函数

闭包的应用除了在装饰器和延迟求值外，还可以利用其特性来定义不同的泛型函数。

【实例 10-4】演示了可以用来进行泛型函数定义的实例，代码如下：

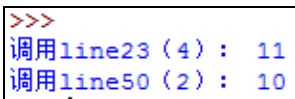
```
def line(a,b):
```

```
def aline(x):
    return a*x + b           #泛型函数中可以使用不同的 a、b 值
    return aline

if __name__ == '__main__':
    line23 = line(2,3)       #a、b 值分别为 2、3 的一次函数
    line50 = line(5,0)       #a、b 值分别为 5、0 的一次函数
    print(line23(4))         #x 为 4 时的第一种类型函数值
    print(line50(2))         #x 为 2 时的第二种类型函数值
```

【代码说明】代码中仍然使用了闭包来进行泛型函数的定义，它可以实现一系列的同类函数。对本例来说，可以实现所有类型的一次函数的求值。

【运行效果】由第一次 a、b 值知道，其一次函数的形式应该为 $2x+3$ ；第二个一次函数的形式应该为 $5x$ 。所以 `line23(4)` 的结果为 $2 \times 4 + 3 = 11$ ，`line50(2)` 的计算结果为 $5 \times 2 = 10$ ，如图 10.4 所示。



```
>>>
调用line23(4): 11
调用line50(2): 10
```

图 10.4 泛型函数调用示例结果

从以上所有关于闭包的应用可以看出，它们都是应用嵌套函数持有定义环境变量的特性来完成功能的。

10.3 上下文管理器



上下文管理器是指实现上下文管理协议方法的对象，它主要用于安全地释放资源（如打开的文件、数据库连接或网络连接、对对象的锁定等）。对于上下文管理器对象，可以使用 `with` 语句来使用它，在 `with` 语句中可以使用上下文管理器管理或提供资源，当退出 `with` 语句时，由上下文管理器来负责安全地释放资源。

上下文管理器的协议方法有以下两个：

```
__enter__(self)
__exit__(self, type, value, tb)
```

`__enter__(self)` 方法是进入上下文时调用的，它创建并返回一个可以引用的资源对象，供 `with` 语句块中的程序使用。

`__exit__(self, type, value, tb)` 方法是退出上下文时调用的，它主要用来安全地释放资源对象。方法中的参数 `type`、`value`、`tb` 用于跟踪退出错误时发生的错误类型、值和跟踪信息。

使用上下文管理器的 `with` 语句的形式为：

```
with context as var:
    pass
```

其中的变量 `var` 将取得上下文管理器的 `__enter__()` 方法所返回的资源引用，供 `with` 后的代码块中使用。

【实例 10-5】演示了自定义的一个上下文管理器及其使用方法，代码如下：

```
class FileMgr:                                     #实现上下文管理协议方法的类

    def __init__(self, filename):                  #构造函数（初始化文件名）
        self.filename = filename
        self.f = None
```



```

def __enter__(self):
    self.f = open(self.filename,encoding='utf-8')
    return self.f
#定义协议方法
#返回资源引用

def __exit__(self,t,v,tb):
    if self.f:
        self.f.close()
#定义协议方法
#释放文件资源

if __name__ == '__main__':
    with FileMgr('a10_4.py') as f:
        for line in f.readlines():
            print(line,end='')
#用上下文管理器直接打开文件
#并返回文件资源引用供操作

```

【代码说明】代码中首先定义了一个管理文件资源对象的上下文管理器类 `FileMgr`，它实现了上下文管理器的协议方法。在进入时打开文件，退出时关闭文件。然后用 `with` 语句来使用这个上下文管理器，打开一个指定的文件，并输出其中的内容。使用了这个管理文件的上下文管理器后，在每次使用文件资源时都不用写打开、关闭文件的代码，就像操作一般变量一样来操作就行了。

【运行效果】如图 10.5 所示，使用上下文管理器打开本章的实例 10-3 文件。

```

>>>
def line(a,b):
    def aline(x):
        return a*x + b
    return aline

if __name__ == '__main__':
    line23 = line(2,3)
    line50 = line(5,0)
    print('调用line23 (4): ',line23(4))
    print('调用line50 (2): ',line50(2))

```

图 10.5 定义和使用上下文管理器

此外，在 Python 标准库中还有一个关于上下文管理器的模块 `contextlib`，其中还有一个可以将生成器转变为上下文管理器的装饰器 `contextmanager`。

【实例 10-6】演示了通过装饰器 `contextmanager` 实现的一个上下文管理器，代码如下：

```

import contextlib

@contextlib.contextmanager
def my_mgr(s,e):
    print(s)
    yield s + ' ' + e
    print(e)
#yield 后的表达式值即为 as 后变量的值

if __name__ == '__main__':
    with my_mgr('start','end') as val:
        print(val)

```

【代码说明】代码中定义了一个用装饰器 `contextlib.contextmanager` 装饰的生成器 `my_mgr()` 方法，使它成为一个上下文管理器。最后使用 `with` 语句来测试这个上下文管理器。

【运行效果】在进入管理器时执行了 `yield` 语句之前的语句，退出 `with` 语句时，执行了 `yield` 语句之后的语句，输出见图 10.6。

```
>>>
start
start end
end
```

图 10.6 使用装饰器生成上下文管理器



上下文管理器的作用是方便资源管理的一种语法形式。

10.4 用字符串操作对象属性

在 Python 的内建函数中有两个函数：`hasattr()`和`setattr()`。

通过`hasattr()`可以测试某个对象是否具有某个属性。它们的原型分别如下。

`hasattr(object, name)`

- **object**: 被测试的对象（类或函数等）；
- **name**: 属性名（字符串格式）。

通过`setattr()`可以设置某个对象属性值，甚至可以添加属性并赋值。

`setattr(object, name, value)`

- **object**: 要设置的对象（类或函数等）；
- **name**: 要设置的属性名（字符串格式）；
- **value**: 要设置的属性值。

【实例 10-7】演示了通过`setattr()`修改对象属性的实例，代码如下：

```
class DemoClass:                                #定义一个用于演示的类
    class_val = 3
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
        self.info()

    def info(self):
        print('类属性 class_val: ', DemoClass.class_val)
        print('实例属性 x: ', self.x)
        print('实例属性 y: ', self.y)

if __name__ == '__main__':
    dc = DemoClass()                             #实例化类
    if hasattr(DemoClass, 'class_val'):
        setattr(DemoClass, 'class_val', 1000)    #设置类属性的值
    if hasattr(dc, 'x'):
        setattr(dc, 'x', 'xxxxxxxxx')           #设置实例发生的值
    if hasattr(dc, 'y'):
        setattr(dc, 'y', 'yyyyyyyyy')
    dc.info()
    setattr(dc, 'z', 'zzzzzzzz')                 #添加并设置实例属性的值
    print('添加的属性 z', dc.z)
```

【代码说明】代码中先定义了一个用于演示的类 `DemoClass`，它具有类属性和实例属性。之后实例化这个类，用`hasattr()`来测试属性，并用`setattr()`来设置其属性的值。



设置类属性时就使用类名进行设置。



【运行效果】如图 10.7 所示，通过 `setattr()` 修改了类属性 `class_val` 的值为 1000。设置实例属性 `x`、`y` 分别为 `xxxxxxx` 和 `yyyyyyy`，还添加了属性 `z`，并同时设置值为 `zzzzzzzz`。

```
>>>
类属性class_val: 3
实例属性x: 0
实例属性y: 0
类属性class_val: 1000
实例属性x: xxxxxxxx
实例属性y: yyyyyyyy
添加的属性z zzzzzzzz
```

图 10.7 用字符串操作类属性的运行结果

这些操作只有在编写框架时或特殊的情况下，才会使用。一般情况下，应该使用面向对象的方法来设置属性。



注意 用字符串操作对象属性，一般在框架的编程中应用比较广泛。在一般的编程中，应尽量通过调用类的方法来操纵类的数据。

10.5 用字典构造分支程序

一般来说，分支程序一般要用 `if` 语句来写，在 Python 语言中，函数也是对象，而且是顶层对象，可以作为参数传入并调用的。因此，可以把它作为数据项放入字典中，并以键值来作为调用（分支）的标志。

【实例 10-8】演示了使用字典来构造分支程序的实例，代码如下：

```
import random
## 以下定义三个分支函数
def path_a():
    print('路径分支 A')

def path_b():
    print('路径分支 B')

def path_c():
    print('路径分支 C')

if __name__ == '__main__':
    path_dict = {}                                #定义空字典，并以键值为标志，把分支函数作为数据项
    path_dict['a'] = path_a
    path_dict['b'] = path_b
    path_dict['c'] = path_c
    paths = 'abc'
    for i in range(4):
        path = random.choice(paths) #从所有的路径中随机选择一个路径
        print('选择了路径: ',path)
        path_dict[path]()           #进入分支
```

【代码说明】代码中定义了三个分支函数 `path_a()`、`path_b()`、`path_c()`，在主程序中构造了一个以键值为标志、把分支函数作为数据项的字典，然后通过键值来确定执行哪个分支函数。

【运行效果】如图 10.8 所示，随机选择路径并执行了该路径分支的函数而输出相关信息。

```
>>>
选择了路径: b
路径分支B
选择了路径: a
路径分支A
选择了路径: c
路径分支C
选择了路径: c
路径分支C
```

图 10.8 用字典做分支的运行结果



用字典构造分支程序虽然可行，但给阅读和理解程序带来麻烦，请慎用。

10.6 重载类的特殊方法

在 Python 中，类有一些以两条下画线开始并且以两条下画线结束的方法，被称为类的专有方法。专有方法是针对类的特殊操作的一些方法，例如在类实例化时将调用 `__init__` 方法。部分类的专有方法及其描述如表 10.1 所示。

表 10.1 类的特殊方法及其描述

方 法 名	描 述
<code>__init__</code>	构造函数，生成对象时调用
<code>__del__</code>	析构函数，释放对象时调用
<code>__add__</code>	加运算
<code>__mul__</code>	乘运算
<code>__cmp__</code>	比较运算
<code>__repr__</code>	打印，转换
<code>__setitem__</code>	按照索引赋值
<code>__getitem__</code>	按照索引获取值
<code>__len__</code>	获得长度
<code>__call__</code>	函数调用

【实例 10-9】演示了自定义的类来实现类的特别的运算方式，代码如下：

```
class Book:                                     #定义一个类 Book

    def __init__(self,name="Python 从入门到精通"):
        self.name = name

    def __add__(self,obj):                       #重载这个方法，实现类相加
        return self.name + ' add ' + obj.name   #得到书名相加

    def __len__(self):                           #重载这个方法，使其求长度
        return len(self.name)                  #得到书名的字符长度

if __name__ == '__main__':
    booka = Book()                              #实例化类
    bookb = Book('Java 从入门到精通')
    print("len(booka):",len(booka))              #求其长度并输出
    print('len(bookb):',len(bookb))
    print(booka + bookb)                        #两个类相加
```



【代码说明】代码中定义了 Book 类，并重载了特殊的方法来实现类的相加和求类的长度。虽然这有点不符合实际，但演示了通过重载类的特殊方法来实现类的运算。

【运行效果】如图 10.9 所示，Book 类是可以求长度的，并且返回的是书名的字符串长度。而 Book 类是可以相加的，就是将两个书名中间用“add”连接起来。

```
>>>
len(booka): 13
len(bookb): 11
Python 从入门到精通 add Java 从入门到精通
```

图 10.9 实现自定义类的自定义运算运行结果

10.7 鸭子类型（duck typing）与多态

Python 语言属于动态类型语言，所以变量的类型是不确定的。鸭子类型（duck typing）是动态类型的一种风格。在这种风格中，一个对象有效的语义不是由继承自特定的类或实现特定的接口来决定，而是由当前方法和属性的集合决定。

鸭子类型概念的名字来源于 James Whitcomb Riley 提出的鸭子测试。“鸭子测试”是这样表述的：

“当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。”

在鸭子类型中，关注的不是对象的类型本身，而是它是如何使用的。例如，可以在不使用鸭子类型的语言中编写一个函数，它接受一个类型为鸭子的对象，并调用它的“走”和“叫”方法。在使用鸭子类型的语言中，这样的函数可以接受一个任意类型的对象，并调用它的“走”和“叫”方法。如果这些需要被调用的方法不存在，那么将引发一个运行时错误。任何拥有类似鸭子“走”和“叫”方法的对象都可被函数接受，这种决定类型的方式因此得名。

所以，在 Python 面向对象的编程中，没有谈及多态，也不用定义接口。在 Python 语言中，调用时是不检查参数类型的，如果被调用的方法不存在，则会引发错误。

【实例 10-10】演示了鸭子类型的使用实例，代码如下：

```
class Duck:
    def __init__(self, name='duck'):
        self.name = name

    def quack(self):
        print("嘎嘎嘎...")

class Cat:
    def __init__(self, name='cat'):
        self.name = name

    def quack(self):
        print("喵喵喵...")

class Tree:
    def __init__(self, name='tree'):
        self.name = name

def duck_demo(obj):
    obj.quack()

if __name__ == '__main__':
    duck = Duck()
```

```

cat = Cat()
tree = Tree()
duck_demo(duck)           #调用参数鸭子类型
duck_demo(cat)            #调用参数鸭子类型
duck_demo(tree)           #会引发错误

```

【代码说明】代码中定义了三个类，前两个类具有 quack() 方法，而后一个类不具有 quack() 方法。在主程序中把它们的实例分别传入函数 duck_demo() 中作为参数，并在其中调用其 quack() 方法。Tree 类实例传入时，会引发错误。

【运行效果】传入前两个类的实例作为参数时，是可以正常运行并输出的。而传入了第三个类的实例时，则由于它不具有 quack() 方法，而引发错误，其输出如图 10.10 所示。

```

>>>
嘎嘎嘎...
喵喵喵...
Traceback (most recent call last):
  File "D:/lx/10/a10_10.py", line 29, in <module>
    duck_demo(tree)
  File "D:/lx/10/a10_10.py", line 21, in duck_demo
    obj.quack()
AttributeError: 'Tree' object has no attribute 'quack'

```

图 10.10 鸭子类型测试的输出结果

在实际的程序设计中，通常不测试方法和函数中参数的类型，而是依赖文档、清晰的代码和测试来确保正确使用。



注意 鸭子类型虽然方便使用，但在编程时应给出明确的说明，防止调用错误。

10.8 小结

本章主要介绍了 Python 语言中比较高阶的技巧、应用和常见的编程方法，其中包含在函数中引用全局变量与命名空间的关系、闭包的基本概念及其基本应用、泛型函数的定义方法，其次介绍了上下文管理器的应用方法与自定义上下文管理器、使用字符串来操作对象属性的方法及用字典构造分支程序。最后介绍了通过重载类的特殊方法实现新数据结构的运算、Python 的动态类型。通过学习本章，你应该掌握这些高阶编程技巧及其应用场合，但要注意不应滥用它们，毕竟这会使程序失去简单易懂的特点。

10.9 本章习题

一、选择题

1. 现有一模块 a_md1，代码如下：

```

x = 3
def mdl_fun(x):
    print(x)
def mdl_fun2():
    print(x)

```

而在主程序中的程序段为：

```

from a_md1 import mdl_fun, mdl_fun2
x = 0
mdl_fun2()
mdl_fun(x)

```




在主程序段函数调用输出值为 ()。

- A. 0 3
- B. 3 0
- C. 0 0
- D. 3 3

【解析】第一次调用时，函数中引用的是其定义模块中的全局变量 `x` 的值 3；而第二次调用时，输出的是提供的本地模块中的变量 `x` 的值为 0。答案为 B。

2. 下列关于闭包说法错误的是 ()。

- A. 闭包可以实现泛型函数
- B. 嵌套函数可以使用其紧邻外层的函数中的变量
- C. 装饰器也是闭包的一种应用
- D. 上下文管理器是闭包的一种应用

【解析】上下文管理器是实现其协议方法的对象，与闭包无关。答案为 D。

3. 有关上下文管理器说法错误的是 ()。

- A. 它要实现 `__enter__()` 和 `__exit__()` 协议方法
- B. 它可以用于安全地释放资源
- C. 它不可以作为一个普通的类使用
- D. 它需要使用 `with` 语句配合使用

【解析】上下文管理器也是一个普通的类。答案为 C。

4. 对以下定义的类使用有错误的是 ()。

```
class Animal:
    def __init__(self, name=""):
        self.name = name
        self.x = 0
    def __add__(self, obj):
        return self.name + obj.name
    def mov(self, x):
        self.x = x
cat = Animal('cat')
dog = Animal('dog')
print(cat + dog)
cat.move(3)
setattr(dog, 'x', 10)
getattr(cat, x)
```

- A. `print(cat + dog)`
- B. `cat.move(3)`
- C. `setattr(dog, x, 10)`
- D. `getattr(cat, x)`

【解析】`getattr` 方法用字符串来操作，这里直接用 `x` 是错误的。答案为 D。

二、实验题

1. 自定义一个实现迭代生成小于某个整数的所有素数的迭代器。
2. 编程实现一个上下文管理器，实现文件使用前打开文件，并确保使用后关闭文件。
3. 尝试用闭包定义一个延迟求值的例子。
4. 用字典构造分支的方法来实现以下用户输入的计算：

```
3 add 5, 结果为 8
5 sub 1, 结果为 4
3 mul 5, 结果为 15
```

第 11 章 文件与文件系统

用编写程序来解决实际项目时，很多时候都离不开文件和文件系统的操作。程序本身就是保存在文件系统中的文件中的。文件既可以保存程序代码，也可以用来保存各种输入与输出数据。文件和文件系统的处理是任何高级程序设计语言必不可少的一部分。

Python 语言提供了丰富的文件操作功能，主要包括用于打开文件的内建函数及标准库中的 OS 包，还有一些文件处理的相关功能模块 fileinput 等。

本章主要介绍 Python 语言中的文件和文件系统的相关操作，内容包括：

- 文件读/写操作；
- 处理文件中的数据；
- fileinput 操作文件；
- 目录操作；
- Python 程序文件打包为 exe。

11.1 文件操作基础



11.1.1 open()函数

在 Python 中可以通过内建的文件打开函数 `open()` 来打开文件，并用相关的方法读/写文件的内容供程序处理和使用，而文件也可以看作 Python 中的一种数据类型。当使用 Python 的内置函数 `open` 打开一个文件后，就返回一个文件对象。其原型如下：

```
open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None,
closefd=True, opener=None)
```

其主要参数的含义如表 11.1 所示。

表 11.1 open()函数参数表

参 数 名	意 义
filename	要打开的文件名
mode	可选参数，文件打开模式
bufsize	可选参数，缓冲区大小
encoding	文件编码类型
errors	编码错误处理方法
newline	控制通用换行符模式的行为
closefd	控制在关闭文件时是否彻底关闭文件

`mode` 是打开文件的操作模式字符串，常用的模式字符及含义如表 11.2 所示。

表 11.2 模式字符含义表

模式字符	表示的操作
r	只读（默认）
w	可写，会清除文件内容



续表

模式字符	表示的操作
a	附加数据
b	二进制数据模式
x	新建一个文件，可写
+	打开文件直接更新
t	文本模式（默认）

这些字符可以组合在一起表示对文件的操作模式，如'rb'表示以只读模式打开二进制文件；'wt'表示以可写模式打开文本文件等。bufsize 为 0 时表示打开文件不用缓冲，为 1 时表示进行缓冲，为负值时则使用系统默认值，为任何其他值则表示使用缓冲的字节数。encoding 表示文件编码的类型，如'gbk'、'utf-8'等，以避免读取文件内容出现乱码。newline 表示换行符模式，因为在不同的系统下，换行符有所不同，换行符主要有'\n'、'\r'、'\r\n'等。

常用的文件操作及其功能如表 11.3 所示。

表 11.3 常用文件操作及其功能

文件操作	功能描述
file.read([n])	将整个文件读入到字符串中，或指定 n 字节
file.readline([n])	读入文件的一行到字符串中
file.readlines()	将整个文件按行读入到列表中
file.write(s)	向文件中写入字符串
file.writelines(lines)	向文件中写入一个行数据列表
file.close()	关闭打开的文件

当 Python 处理文件中的数据时，可以使用 while 语句来循环读取文件中的行：

```
while True:
    line = f.readline()
    if not line:
        break
```

也可以用 for 语句来迭代文件中的所有行：

```
for line in f:
    pass
```

【实例 11-1】 演示了一个处理文件中数据的例子，代码如下：

```
def file_hdl(name='python.txt'):          #定义一个文件数据处理函数
    f = open(name)                        #打开文件
    res = 0                               #累加器变量
    i = 0                                 #计算读取的行数
    for line in f:                         #迭代文件中的行
        i += 1
        print('第%s行的数据为: ' % line.strip(),line)
        res += int(line)                  #累加数据

    print('这些数的和为: ',res)

    f.close()                             #关闭文件

if __name__ == '__main__':
    file_hdl()                            #调用函数
```

【代码说明】代码中只定义了一个函数，其功能是打开一个文件，迭代处理文件中的每一行数据，并把每一行数据转换为一个整数，然后累加起来输出。

【运行效果】程序的运行效果如图 11.1 所示。

```
>>>
第1行的数据为: 1
第2行的数据为: 2
第3行的数据为: 3
第4行的数据为: 4
第5行的数据为: 5
这些数的和为: 15
```

图 11.1 程序的运行结果

在处理文件之前要打开文件，处理结束后还要关闭文件。在 Python 中可以使用 with 语句来管理文件的打开和关闭。例如实例 11-1 中的函数可改写为：

```
def file_hdl(name='python.txt'):          #定义一个文件数据处理函数
    with open(name) as f:                 #打开文件
        res = 0                           #累加器变量
        i = 0                             #计算读取的行数
        for line in f:                   #迭代文件中的行
            i += 1
            print('第%s行的数据为: ' % line.strip(),line)
            res += int(line)              #累加数据

    print('这些数的和为: ',res)
```

如果向文件中写入数据，打开模式一定要是可写的，并要注意是覆盖文件中的数据，还是附加数据到文件中。写入文件的相关操作和读取基本上相同，这里就不再举例说明了。



注意 当需要打开的文件很大时，可能会占用大量的内存，你可以将其分割后打开。

11.1.2 用 fileinput 操作文件

fileinput 模块提供了一个以行模式循环处理一个或多个文件内容的功能，它实现了对文件中行的“懒惰”迭代，读取时不需要把文件内容放入内存，这样可以提高程序的效率。

fileinput 模块中常用的函数有：

- input(): #返回能够用于迭代一个或多个文件中所有行的对象；
- filename(): #返回当前文件的名称；
- lineno(): #返回当前读取的行的数量；
- isfirstline(): #返回当前行，判断是否是文件的第一行，是当前文件的第一行时返回真值，反之则返回假值；
- filelineno(): #返回当前读取行在文件中的行数。

fileinput 也支持上下文管理器，可以使用 with 语句来进行操作，而不用在使用后手工关闭对象。

【实例 11-2】演示了该模块的基本使用方法，代码如下：

```
import fileinput
def demo_fileinput():
```



```

with fileinput.input(['fpa.txt','fpb.txt']) as lines:    #用with 语句
    for line in lines:                                  #迭代处理两个文件
        print("总第%d行," % fileinput.lineno(),
              "文件%s 中第%d行: " %
              (fileinput.filename(),fileinput.filelineno()))
        print(line.strip())

if __name__ == '__main__':
    demo_fileinput()

```

【代码说明】代码中首先导入了 `fileinput` 模块，在随后定义的普通函数中，使用 `fileinput.input()` 函数来迭代处理两个文本文件（'fpa.txt','fpb.txt'），以列表形式提供给 `input()` 函数作为参数。最后迭代处理显示每行，同时输出每行的行号。

【运行效果】如图 11.2 所示，运行时显示了所有文件中的行号及每一行的内容。

```

>>>
总第1行, 文件fpa.txt中第1行:
fpa-1
总第2行, 文件fpa.txt中第2行:
fpa-2
总第3行, 文件fpa.txt中第3行:
fpa-3
总第4行, 文件fpb.txt中第1行:
fpb-1
总第5行, 文件fpb.txt中第2行:
fpb-2
总第6行, 文件fpb.txt中第3行:
fpb-3

```

图 11.2 用 `fileinput` 迭代显示多个文件内容

11.2 常用文件和目录操作

在计算机系统中进行操作时，免不了要与文件和目录打交道。对一些比较烦琐的文件和目录操作，可以使用 Python 提供的 OS 模块来进行。OS 模块中包含很多操作文件和目录的函数，可以方便地进行重命名文件、添加/删除目录、复制目录/文件等操作。

11.2.1 获得当前路径

在 Python 中可以使用 `os.getcwd()` 函数获得当前的路径。其原型如下：

```
os.getcwd()
```

该函数不需要传递参数，它返回当前的目录。需要说明的是，当前目录并不是指程序所在的目录，而是所运行程序的目录。

在交互式环境下，函数运行的示例如下：

```

>>> import os
>>> print("当前目录是:",os.getcwd())
当前目录是: D:\lx\11

```

11.2.2 获得目录中的内容

在 Python 中可以使用 `os.listdir()` 函数获得指定目录中的内容。其原型如下：

```
os.listdir(path)
```

`path` 是要获得内容目录的路径，在交互式环境下获得当前目录的内容如下：

```

>>> os.listdir()
['all_1.py', 'all_2.py', 'fpa.txt', 'fpb.txt', 'python.txt', 'test']

```

可以看出, 当前目录中的文件和目录的名称是以列表形式返回的。

11.2.3 创建目录

在 Python 中可以使用 `os.mkdir()` 函数创建目录。其原型如下:

```
os.mkdir(path)
```

其参数含义如下。

path: 要创建目录的路径。

以下在交互式环境中实现在当前目录下创建 `test2` 目录, 之后显示的内容中已经创建了目录:

```
>>> os.mkdir('test2')
>>> os.listdir()
['all_1.py', 'all_2.py', 'fpa.txt', 'fpb.txt', 'python.txt', 'test', 'test2']
```

11.2.4 删除目录

在 Python 中可以使用 `os.rmdir()` 函数删除目录。其原型如下:

```
os.rmdir(path)
```

其参数含义如下。

path: 要删除的目录的路径。

以下在交互式环境下实现删除当前目录下的 `test2` 目录:

```
>>> os.rmdir('test2')
>>> os.listdir()
['all_1.py', 'all_2.py', 'fpa.txt', 'fpb.txt', 'python.txt', 'test']
```



说明 使用 `os.rmdir` 删除的目录必须为空目录, 否则函数出错。如果删除的目录不存在, 也会报错。

11.2.5 判断是否是目录

在 Python 中可以使用 `os.path.isdir()` 函数判断某一路径是否为目录。其函数原型如下:

```
os.path.isdir(path)
```

其参数含义如下。

path: 要进行判断的路径。

以下在交互式环境下实现目录判断:

```
>>> os.path.isdir('test')
True          # 为 True 表示是目录, 为 False 则表示不是目录或不存在
>>> os.path.isdir('fpb.txt')
False         # 为 False 则表示不是目录或不存在
```

11.2.6 判断是否为文件

在 Python 中可以使用 `os.path.isfile()` 函数判断某一路径是否为文件。其函数原型如下:

```
os.path.isfile(path)
```

其参数含义如下。

path: 要进行判断的路径。

它与 `os.path.isdir()` 的用法相同, 这里就不举例说明了。



注意 在进行目录操作时，需要具有相应的权限，否则会导致错误。

11.2.7 遍历某目录下的所有文件和目录

在 Python 中可以使用 `os.walk()` 函数，其可以递归地遍历指定目录下的所有文件和子目录，而 `os.walk()` 函数返回的是一个可以迭代的生成器，要处理遍历得到的结果，可以使用 `for` 语句来循环处理。其函数原型如下：

```
os.walk(path)
```

其参数含义如下。

path: 要进行遍历的目录路径。

以下在交互式环境下实现遍历当前目录并输出：

```
>>> for i in os.walk('.\\'):
    print(i)

('.\\', ['test'], ['all_1.py', 'all_2.py', 'fpa.txt', 'fpb.txt', 'python.txt'])
('.\\test', ['testsub'], [])
('.\\test\\testsub', [], [])
```

通过以上输出可以看出，`os.walk()` 遍历当前目录后输出的格式为多个元组，每个元组的第一项为遍历的目录名（字符串），第二项为遍历目录中的子目录列表，第三项为遍历目录中所有文件的列表。

此外，还有两个代表当前平台下的行分隔符（`os.linesep`）和目录名分隔符（`os.pathsep`），可以在不同的平台下正确运行，而不会使程序出错。

11.2.8 由文件名批量获取姓名和考号

在教学中，经常要求学生以文件方式提交作品，以供评分或评比。一般来说会要求提交的文件包括姓名和序号（考号），当学生提交了所有的文件后，你必须获取其姓名和考号，放入电子表格中进行处理或打分。那么，你可以通过以下程序完成这个简单的任务。

【实例 11-3】 演示了由文件名批量获取姓名和考号，代码如下：

```
import os

filenames = []                                #所有文件名的存放列表

for a,b,files in os.walk('test'):             #获取当前目录下 test 目录中的所有文件
    if files:
        filenames.append([file[:-4] for file in files])#扩展名为 3 个字母

fname = 'testexam'                            #指定生成电子表格的文件名
i = 0
for files in filenames:
    f=open(fname+str(i)+'.xls','w')            #打开指定的文件
    for name in files:
        f.write(name[-2:]+'\t'+name[:-2]+'\n') #写入姓名和考号
    f.close()                                  #关闭文件
    i += 1
print("成功生成！")
```

【代码说明】 代码中通过 `os.walk()` 对目录下的所有文件进行遍历，获取包含学生信息的所有文件名字符串放入列表 `filenames` 中，根据指定的电子表格文件名将学生姓名和考号写入文

件中。

【运行效果】假设学生上交的文件有两个，如图 11.3 所示，则会批量自动生成考号和姓名到电子表格文件中。

	A	B
1	12	周兰巧
2	34	李文

图 11.3 目录中文件名和提取的信息

11.2.9 批量文件重命名

在日常工作中经常会遇到这样的情况，需要将某个文件夹下的文件按照一定的规则重新命名。如果用手工作方式将文件逐个重命名，需要耗费大量的时间，而且操作过程容易出错。在学习了 Python 以后，完全可以写一个简单的程序完成这样的工作。

【实例 11-4】演示了批量重命名，代码如下：

```
import os
perfix = 'Python'                # perfix 为重命名后的文件起始字符
length = 2                      # length 为除去 perfix 后，文件名要达到的长度
base = 1                        # 文件名的起始数
format = 'mdb'                  # 文件的后缀名
# 函数 PadLeft 将文件名补全到指定长度
# str 为要补全的字符
# num 为要达到的长度
# padstr 未达到长度所添加的字符
def PadLeft(str, num, padstr):
    stringlength = len(str)
    n = num - stringlength
    if n >= 0:
        str = padstr * n + str
    return str
# 为了避免误操作，这里先提示用户
print('the files in %s will be renamed' % os.getcwd())
all_files = os.listdir(os.getcwd())
print([f for f in all_files if os.path.isfile(f)]) # 输出当前目录下的所有文件名
input = input('press y to continue\n') # 获取用户输入
if input.lower() != 'y':                # 判断用户输入，以决定是否执行重命名操作
    exit()
filenames = os.listdir(os.getcwd())     # 获得当前目录中的内容
# 基数减 1，为了下面 i = i + 1 在第一次执行时等于基数
i = base - 1
for filename in filenames:              # 遍历目录中的内容，进行重命名操作
    i = i + 1
    # 判断当前路径是否为文件，并且不是“rename.py”
    if filename != "rename.py" and os.path.isfile(filename):
        name = str(i)                  # 将 i 转换成字符
        name = PadLeft(name, length, '0') # 将 name 补全到指定长度
        t = filename.split('.')         # 分割文件名，以检查其是否是所要修改的类型
        m = len(t)
        if format == '':                # 如果未指定文件类型，则更改当前目录中的所有文件
            os.rename(filename, perfix + name + '.' + t[m-1])
        else:                           # 否则只修改指定类型
            if t[m-1] == format:
```




```

        os.rename(filename, prefix+name+'.'+t[m-1])
    else:
        i = i - 1                # 保证 i 连续
    else:
        i = i - 1                # 保证 i 连续
all_files = os.listdir(os.getcwd())
print([f for f in all_files if os.path.isfile(f)]) #输出当前目录下所有文件名

```

【代码说明】代码运行后会输出当前目录下的所有文件，并将扩展名为 mdb 的文件批量进行命名，其他类型的文件不做处理。

【运行效果】如图 11.4 所示，命名前，当前目录下有四个文件，只有两个扩展名为 mdb 的文件（ad.mdb、vf.mdb），当接收输入 y 允许后，对文件名进行了批量处理。最后一行显示两个文件分别被命名为 Python01.mdb 和 Python02.mdb。

```

D:\lx\11\test>python a11_3.py
the files in D:\lx\11\test will be renamed
['a11_3.py', 'ad.mdb', 'python.txt', 'vf.mdb']
press y to continue
y
['a11_3.py', 'python.txt', 'Python01.mdb', 'Python02.mdb']

```

图 11.4 对文件批量重命名

11.3 编译为可执行文件

11.3.1 用 py2exe 生成可执行程序

目前，py2exe 0.9.2 版本已经支持 Python 3.x，它可以将 Python 程序打包为 Windows 下独立的可执行文件。

要使用 py2exe，首先要编写一个编译程序（例如编写一个名为 setup.py 的程序），然后在 Python 中运行编译 setup.py，即可将需要封装的其他 Python 程序编译成可执行文件。

【实例 11-5】演示了使用 py2exe 来封装 Python 程序为可执行文件，代码如下：

```

#文件 hello.py                                #待封闭程序文件
import ctypes                                  #导入包
print('Hello,Python!')

#文件 setup.py                                #编译程序
from distutils.core import setup
import py2exe

setup(console=['hello.py'])                    #指定为控制台程序的主程序文件名

然后，在命令提示符下执行以下命令就可以完成打包：
python setup.py py2exe

```



注意 打包后的文件默认放入当前目录下的 dist 子目录中。

【代码说明】本例演示了将本书第一个 Python 程序（最简单的 hello.py 程序）打包为 exe 文件的实例。打包成功后进入编译目录，执行对应的 exe 文件即可得到运行结果。

代码中编译的语句是 setup(console=['hello.py'])，方括号中的就是要编译的程序名，前面的 console 表示将其编译成命令行界面程序。如果要编译 GUI 的可执行文件，则将 console 改为 Windows。另外，如果需要将程序编译成 Windows 服务，则可以使用 service 选项。

【运行效果】打包的过程、输出信息和打包后直接在控制台执行对应可执行文件的输出结果如图 11.5 所示。

```
D:\lx\11\exe>python setup.py py2exe
running py2exe

 3 missing Modules
-----
? readline                imported from cmd, code, pdb
? win32api                imported from platform
? win32con                imported from platform
Building 'dist\hello.exe'.
Building shared code archive 'dist\library.zip'.
Copy c:\windows\system32\python34.dll to dist
Copy D:\Python34\DLLs\select.pyd to dist\select.pyd
Copy D:\Python34\DLLs\ssl.pyd to dist\ssl.pyd
Copy D:\Python34\DLLs\pyexpat.pyd to dist\pyexpat.pyd
Copy D:\Python34\DLLs\lzma.pyd to dist\lzma.pyd
Copy D:\Python34\DLLs\bz2.pyd to dist\bz2.pyd
Copy D:\Python34\DLLs\ctypes.pyd to dist\ctypes.pyd
Copy D:\Python34\DLLs\unicodedata.pyd to dist\unicodedata.pyd
Copy D:\Python34\DLLs\hashlib.pyd to dist\hashlib.pyd
Copy D:\Python34\DLLs\socket.pyd to dist\socket.pyd

D:\lx\11\exe>cd dist
D:\lx\11\exe\dist>hello
Hello,Python!
```

图 11.5 py2exe 打包程序及运行结果



注意

当打包后运行程序出现无法导入某个模块或找不到某个模块时，可以直接在程序中添加相应的 import 语句，对其进行导入即可。

例如本例中如果在 hello.py 中不加入 import ctypes 语句，在打包后运行则会出现找到模块 ctypes。

11.3.2 用 cx_freeze 生成可执行文件

与使用 py2exe 类似，首先需将 cx_freeze 下载到本机并在安装后才能使用。下载地址为 <http://sourceforge.net/projects/cx-freeze/files/4.3.3/>，其网页界面如图 11.6 所示。

Looking for the latest version? Download cx_Freeze-4.3.3.win-amd64-py2.7.msi (540.7 kB)			
Home / 4.3.3			
Name	Modified	Size	Downloads / Week
↑ Parent folder			
cx_Freeze-4.3.3-1.src.rpm	2014-05-04	61.4 kB	8 <input type="checkbox"/> ⓘ
cx_Freeze-4.3.3.tar.gz	2014-05-04	59.0 kB	51 <input type="checkbox"/> ⓘ
cx_Freeze-4.3.3.win-amd64-py3.4.msi	2014-05-04	917.5 kB	39 <input type="checkbox"/> ⓘ
cx_Freeze-4.3.3.win-amd64-py3.3.msi	2014-05-04	966.7 kB	11 <input type="checkbox"/> ⓘ
cx_Freeze-4.3.3.win-amd64-py2.7.msi	2014-05-04	540.7 kB	59 <input type="checkbox"/> ⓘ
cx_Freeze-4.3.3.win32-py3.4.msi	2014-05-04	917.5 kB	38 <input type="checkbox"/> ⓘ
cx_Freeze-4.3.3.win32-py3.3.msi	2014-05-04	966.7 kB	14 <input type="checkbox"/> ⓘ
cx_Freeze-4.3.3.win32-py2.7.msi	2014-05-04	704.5 kB	24 <input type="checkbox"/> ⓘ

图 11.6 cx_Freeze 下载界面

下载时需注意选择对应的 Python 版本，以及操作系统平台。例如，在 64 位 Windows 7 的 Python 3.4 中使用，则可下载安装文件 cx_Freeze-4.3.3.win-amd64--py3.4.msi。

安装方法很简单，双击下载的安装文件，然后根据向导单击鼠标就可完成安装。安装完成



后在 C:\Python34\Scripts 目录中可看到 cx_freeze 的相关文件。在命令窗口切换到 C:\Python34\Scripts 目录, 输入以下命令可看到如图 11.7 所示的效果, 表示 cx_freeze 安装成功。

```
D:\Python34\Scripts>cxfreeze -h
Usage: cxfreeze [options] [SCRIPT]

Freeze a Python script and all of its referenced modules to a base
executable which can then be distributed without requiring a Python
installation.

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -O                    optimize generated bytecode as per PYTHONOPTIMIZE; use
                        -OO in order to remove doc strings
  -c, --compress        compress byte code in zip files
  -s, --silent          suppress all output except warnings and errors
  --base-name=NAME      file on which to base the target file; if the name of
                        the file is not an absolute file name, the
                        subdirectory bases (rooted in the directory in which
                        the freezer is found) will be searched for a file
                        matching the name
  --init-script=NAME    script which will be executed upon startup; if the
                        name of the file is not an absolute file name, the
```

图 11.7 cx_freeze 命令行运行界面

将 cx_freeze 安装成功后, 接下来就可进行 Python 程序的编译操作。仍然以上节操作的 hello.py 为例, 演示在 Python 3 中使用 cx_freeze 编译为 exe 文件的方法。

(1) 在命令窗口切换到“hello.py”所在的目录(需打包文件所在目录)。

(2) 输入以下命令:

```
d:\python34\Scripts\cxfreeze --target-dir=dist_cxfreeze hello.py
```

其中的 hello.py 是需要编译的程序文件, dist_cxfreeze 是目标文件夹, 打包后会生成 dist_cxfreeze 目录, 在这个目录中生成编译后的可执行文件。

执行以上命令后, 将显示如图 11.8 所示的输出(部分)。

```
D:\1x\11\exe>d:\python34\Scripts\cxfreeze --target-dir=dist_cxfreeze hello.py
creating directory dist_cxfreeze
copying D:\Python34\lib\site-packages\cx_Freeze\bases\Console.exe -> dist_cxfree
ze\hello.exe
copying C:\Windows\system32\python34.dll -> dist_cxfreeze\python34.dll
writing zip file dist_cxfreeze\hello.exe

Name                               File
-----
m __main__                         hello.py
m _bootlocale
m _bz2                             D:\Python34\DLLs\_bz2.pyd
m _codecs
```

图 11.8 cxfreeze 命令编译 hello.py 命令行界面

最后在当前目录下的 dist_cxfreeze 目录中生成可执行的 exe 文件, 在命令提示符下运行结果与 py2exe 编译运行输出是相同的。

如果想在编译 GUI 界面程序运行时不出现控制台窗口的背景, 则在编译时还需添加一个 base-name 的参数, 具体命令如下:

```
c:\python34\Scripts\cxfreeze gui_filename.py --target-dir dist --base-name=
Win32GUI
```

此外, 比较 py2exe 和 cxfreeze 编译后的目录内容(如图 11.9 所示)可以看出, py2exe 编译出的文件多一点, 体积也稍大(本程序编译的结果显示: py2exe 编译结果为 12 个文件, 体积为 11MB; cxfreeze 编译结果为 4 个文件, 体积约为 5.8MB。)

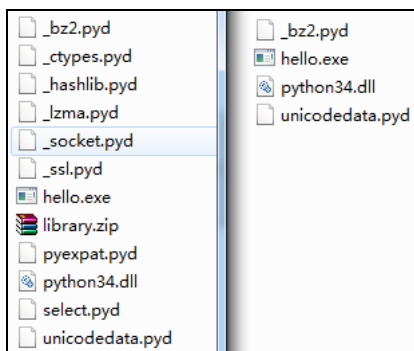


图 11.9 py2exe 和 cxfreeze 编译结果比较 (左为 py2exe, 右为 cxfreeze)

本来 Python 源程序只有一行代码, 编译成的可执行文件包却很大。这主要是因为在没有安装 Python 的系统中, 必须将 Python 的相关库打包进来, 所以其体积较大。

11.4 小结

本章主要介绍了 Python 语言中如何操作文件和目录、处理文件中数据, 以及应用 py2exe 和 cx_freeze 将 Python 程序文件编译为可独立运行的可执行文件。通过学习本章内容, 你应掌握 open() 函数的使用、处理文件中数据、使用常用目录操作函数操作目录, 以及将 Python 程序文件编译为可独立运行的可执行文件。

11.5 本章习题

一、选择题

1. 以下关于 open() 函数说法错误的是 ()。
 - A. 调用时, 可以不用任何参数
 - B. 可以打开一个只读的文件
 - C. 可以实现打开不存在的文件就创建它
 - D. 可以指明是否使用缓冲

【解析】调用时至少得提供要打开的文件名的一个参数。答案为 A。

2. 在 Python 中若要打开一个大文件, 适当的方法是 ()。
 - A. 用 fileinput.fileinput 来打开它
 - B. 用 open() 直接打开
 - C. 用 open() 打开并指定大的缓冲区
 - D. 用只读的方式打开

【解析】fileinput.fileinput 是懒惰迭代打开, 很节约内存。答案为 A。

二、实验题

1. 现有一个文本文件, 其内容是每行包含一名学员的序号、姓名、身份证号、家庭住址, 并且以一个空格分开, 而 pho 文件夹下有以身份证号命名的个人照片, 但其中的照片很多, 也有可能某学员没有照片。请编程实现将这个文本文件中所包含的学员的照片重新命名为学员姓名, 以方便检查并排序。
2. 尝试将以上程序分别用 py2exe 工具和 cx_freeze 工具生成可执行文件。

第 12 章 基于 tkinter 的 GUI 编程

Windows 的图形用户界面非常方便用户操作，因此，Windows 操作系统得到了广大个人计算机用户的欢迎。

在 Python 中，也可以编写美观的 GUI 界面应用程序与项目。tkinter 是 Python 自带的用于 GUI 编程的模块，tkinter 是对图形库 TK 的封装。tkinter 是跨平台的，这意味着在 Windows 下编写的程序，可以不加修改地在 Linux、UNIX 等系统下运行。因此，tkinter 的优势在于其可移植性。

本章主要介绍 Python 标准库中 tkinter 库的 GUI 编程，内容包括：

- GUI 概述；
- tkinter 概述；
- 使用 tkinter 组件；
- 处理 tkinter 组件的事件；
- 使用 tkinter 标准对话框；
- 创建自定义对话框。

12.1 GUI 概述

GUI 是指图形化用户界面，本书前面所有章节中的实例全部是非图形化的用户界面，程序运行时需要的输入、输出及运行环境都是在命令提示符下完成的。本节主要介绍图形化用户界面的基本概念及相关的 Python 库。



12.1.1 GUI 是什么

GUI 是图形用户界面（Graphical User Interface）的英文简称，是指采用图形方式显示的计算机操作用户界面。

在图形用户界面中，计算机画面上显示窗口、图标、按钮等图形，表示不同目的的动作，用户通过鼠标等指针设备进行选择。

此外，现代触摸屏图形用户界面也开始发展起来了。即在用户界面下，直接用手指或者特殊的笔端触摸触摸屏上显示的按钮、图标进行各种操作已经非常普及，如自动取款机、汽车导航、媒体播放器、游戏机等，操作通常简捷、直观。苹果公司的 iPhone 手机还装有支持多点触控的操作系统。

在进行 GUI 程序设计中，不仅要设计业务数据处理程序，还要设计用户界面如窗口、图标、按钮等各种元素的排列、颜色、显示方式等，并将它们连接起来，最终完成用户任务的应用程序。从程序员的角度来说，加大了程序设计的难度；而从用户的角度来说，降低了使用门槛，使得普通用户可以享受程序的使用便利性。

12.1.2 Python 编写 GUI 程序库

Python 语言一诞生，就有不少优秀的 GUI 库整合到其中。常用的 Python 语言的 GUI 库主要有以下 4 种。

1. tkinter

tkinter 是绑定了 Python 的 TkGUI 工具集,就是 Python 包装的 Tcl 代码,通过内嵌在 Python 解释器内部的 Tcl 解释器实现。目前,它是 Python 标准库的一部分,所以使用它进行 GUI 编程是不需要另外安装第三方库的。

2. wxPython

wxPython 是 Python 对跨平台的 GUI 工具集 wxWidgets (C++ 编写)的包装,作为 Python 的一个扩展模块实现。它也是比较流行的一个 tkinter 的替代品,在各种平台下的表现都挺好。但是,目前的版本尚不支持 Python 3.4 版本。

3. PyQt

PyQt 是 Python 对跨平台的 GUI 工具集 Qt 的包装,实现了约 440 个类以及约 6000 个函数或者方法,PyQt 是作为 Python 的插件实现的。其功能非常强大,用 PyQt 开发的界面效果与用 Qt 开发的界面效果相同,其跨平台的支持很好。

4. PySide

PySide 是另一个 Python 对跨平台的 GUI 工具集 Qt 的包装,捆绑在 Python 中,最初由 BoostC++库实现,后来迁移到 Shiboken。

此外,还有一些其他的 GUI 库,如 PyGTK、AnyGui 等。

习惯用 Windows 的话可选择一种在 NET 和 Mono 上实现的 Python 语言——IronPython。它可以调用 .NET 的丰富程序库来编写 GUI 程序。

12.2 tkinter 图形化库简介

使用 tkinter 可以创建完整的 GUI 程序。在 tkinter 中,可以直接使用文本框、按钮、标签等组件(widget)进行 GUI 编程。换句话说,要实现某个界面元素,只要调用对应的组件即可。

tkinter 是 Python 的一个模块,可以像其他模块一样在 Python 的交互式 shell 中(或者“.py”程序中)被导入,tkinter 模块被导入后即可使用 tkinter 模块中的函数、方法等。

12.2.1 创建 GUI 程序第一步

使用 tkinter 创建图形界面时要首先导入 tkinter 模块。可以在 Python 的交互式环境中输入以下语句验证 Python 是否安装了 tkinter 模块:

```
import tkinter
```

如果上述语句执行成功,则表示已经安装了 tkinter 模块。在编写程序时只要使用 import 语句导入 tkinter 模块,即可使用 tkinter 模块中的函数、对象等进行 GUI 编程。

在使用 tkinter 模块时,首先要使用 tkinter.Tk 生成一个主窗口对象,然后才能使用 tkinter 模块中其他的函数、方法等。当生成主窗口以后,可以向其添加组件,或者直接调用其 mainloop 方法进行消息循环。

【实例 12-1】演示了仅创建一个简单的窗口而没有使用组件,代码如下:

```
# -*- coding:utf-8 -*-
import tkinter                                # 导入 tkinter 模块
root = tkinter.Tk()                          # 生成 root 主窗口
root.mainloop()                             # 进入消息循环
```

【代码说明】代码首先导入 tkinter 库,然后生成 root 主窗口,并进入消息循环。



【运行效果】如图 12.1 所示，运行后出现一个主窗口，该窗口具有一般应用程序窗口的基本功能，可以最小化、最大化、关闭，还具有标题栏，甚至可以使用鼠标调整其大小。

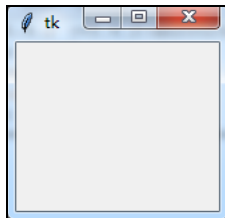


图 12.1 基本的主窗口

该实例仅使用 3 条语句就创建了一个简单的 GUI 窗口。当完成窗口内部组件的创建工作后，也要进入消息循环中，以处理窗口及其内部组件的事件。

如果需要在 tkinter 的窗口、组件中显示中文，除了在“.py”程序文件中的首行添加“#-*-coding:utf-8 -*-”指明字符编码以外，还应该将程序保存成“UTF-8”的编码格式。如果使用记事本，则可以单击“文件”→“另存为”命令，选择“另存为”对话框中“编码”下拉框中的“UTF-8”选项。然后单击“保存”按钮，即可在 tkinter 的窗口或组件中使用中文。

12.2.2 创建 GUI 程序第二步

在 tkinter 中，组件与主窗口一样，也是使用 tkinter 模块中相应的组件函数生成的。组件生成后就可以使用 pack、grid 或 place 方法将其添加到窗口中。

【实例 12-2】演示了具有标签和按钮组件的主窗口，代码如下：

```
# -*- coding:utf-8 -*-
#
import tkinter                                # 导入tkinter 模块
root = tkinter.Tk()                          # 生成root 主窗口
label= tkinter.Label(root, text="Hello, tkinter!") # 生成标签
label.pack()                                # 将标签添加到 root 主窗口
button1 = tkinter.Button(root, text="Button1") # 生成button1
button1.pack(side=tkinter.LEFT)              # 将button1 添加到root 主窗口
button2 = tkinter.Button(root, text="Button2") # 生成button2
button2.pack(side=tkinter.RIGHT)              # 将button2 添加到 root 主窗口
root.mainloop()                              # 进入消息循环
```

【代码说明】代码中直接实例化 tkinter 库中的一个标签（Label）组件和两个按钮组件（Button），然后调用其 pack()方法，将它们添加至主窗口中。

【运行效果】如图 12.2 所示，运行后的主窗口中显示了一个标签（Hello,tkintr!）和两个按钮（Button1,Button2）。

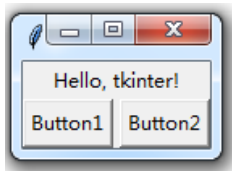


图 12.2 向窗口中添加组件

在使用 tkinter 向窗口中添加组件时，如果不指定组件的位置信息，tkinter 将自动把组件安排在合适的位置。当然，对组件的位置也可以通过精确控制达到需要的效果。

上述程序使用了标签和按钮组件。除此以外，tkinter 还提供了其他的组件，用以满足不同的需求。



注意

运行以上实例后，单击两个按钮均无反应，这是因为在程序中未添加按钮组件单击事件的处理函数。

关于组件的事件处理，将在本章以后的内容中进行讲解，此处仅给出一个简单的创建窗口的例子。

12.3 tkinter 组件

在上一节中创建的窗口实际上是存放组件的一个“容器”。如果仅创建一个不包含组件的窗口，其作用也仅是测试 tkinter 模块。更有意义的做法是，当窗口创建好以后，应根据程序的功能向窗口中添加合适的组件，然后定义与实际相关的处理函数，这样才算一个完整的 GUI 程序。

12.3.1 组件分类

tkinter 中包含了 15 种核心组件，用以实现不同的功能。详细的组件如表 12.1 所示。

表 12.1 tkinter 中的组件名称及其功能

组件名称	组件功能
Button	按钮
Canvas	绘图形组件，可以在其中绘制图形
Checkbutton	复选框
Entry	文本框（单行）
Frame	框架，将几个组件组成一组
Label	标签，可以显示文字或者图片
Listbox	列表框
Menu	菜单
Menubutton	Menubutton 的功能完全可以使用 Menu 替代
Message	与 Label 组件类似，但是可以根据自身大小将文本换行
Radiobutton	单选框
Scale	滑块
Scrollbar	滚动条
Text	文本框（多行）
Toplevel	用来创建子窗口容器组件

12.3.2 布局组件

在前面的例子中仅使用组件的 pack 方法将组件添加到窗口中，而未设置组件的位置。因此，前面例子中组件位置都是由 tkinter 模块自动确定的。

对于包含较多组件的窗口，为了让组件布局合理，可以通过向 pack 传递参数来设置组件在窗口中的位置。除了组件的 pack 方法以外，还可以使用 grid 方法和 place 方法来放置组件。

组件的 pack 方法可以使用以下几个参数来设置组件的位置属性。

- after: 将组件置于其他组件之后；



- anchor: 组件的对齐方式，顶对齐“n”、底对齐“s”、左对齐“w”、右对齐“e”；
- before: 将组件置于其他组件之前；
- side: 组件在主窗口的位置，可以为“top”“bottom”“left”“right”。

组件的 grid 方法使用行列的方法放置组件的位置。组件的 grid 方法可以使用以下几个参数设置组件的位置属性。

- column: 组件所在的列起始位置；
- columnspan: 组件的列宽；
- row: 组件所在的行起始位置；
- rowspan: 组件的行宽。

组件的 place 方法相对较灵活，可以直接使用坐标来放置组件的位置。组件的 place 方法可以使用这些参数设置组件的位置属性，如表 12.2 所示。

表 12.2 place 的参数表

参 数 名	作 用
anchor	组件对齐方式
x	组件左上角的 x 坐标
y	组件左上角的 y 坐标
relx	组件相对于窗口的 x 坐标，应为 0~1 之间的小数
rely	组件相对于窗口的 y 坐标，应为 0~1 之间的小数
width	组件的宽度
height	组件的高度
relwidth	组件相对于窗口的宽度，应为 0~1 之间的小数
relheight	组件相对于窗口的高度，应为 0~1 之间的小数

12.4 常用 tkinter 组件

tkinter 库中有很多 GUI 组件，其中包括在图形化界面中常用的按钮、标签、文本框、菜单、单选框、复选框等，本节主要介绍各种常用组件的使用方法。

12.4.1 按钮

使用 tkinter.Button 时，向其传递参数可以控制按钮的属性，例如，可设置按钮上文本的颜色、按钮的颜色、按钮的大小以及按钮的状态等。常用的控制参数及其作用如表 12.3 所示。

表 12.3 按钮的常用控制参数表

参 数 名	作 用
anchor	指定按钮上文本的位置
background (bg)	指定按钮的背景色
bitmap	指定按钮上显示的位图
borderwidth (bd)	指定按钮边框的宽度
command	指定按钮消息的回调函数
cursor	指定鼠标移动到按钮上的指针样式
font	指定按钮上文本的字体
foreground (fg)	指定按钮的前景色

续表

参 数 名	作 用
height	指定按钮的高度
image	指定按钮上显示的图片
state	指定按钮的状态
text	指定按钮上显示的文本
width	指定按钮的宽度



注意

一些组件具有相似的控制参数，所以你不必强记它。

【实例 12-3】演示了在主窗口中创建各种不同的按钮，代码如下：

```
# -*- coding:utf-8 -*-
#
import tkinter                                # 导入tkinter 模块
root = tkinter.Tk()
button1 = tkinter.Button(root,
                           anchor = tkinter.E,    # 指定文本对齐方式
                           text = 'Button1',      # 指定按钮上的文本
                           width = 40,           # 指定按钮的宽度，相当于 40 个字符
                           height = 5)          # 指定按钮的高度，相当于 5 行字符
button1.pack()                                  # 将按钮添加到窗口
button2 = tkinter.Button(root,
                           text = 'Button2',
                           bg = 'blue')          # 指定按钮的背景色
button2.pack()
button3 = tkinter.Button(root,
                           text = 'Button3',
                           width = 14,           # 指定按钮的宽度
                           height = 1)          # 指定按钮的高度
button3.pack()
button4 = tkinter.Button(root,
                           text = 'Button4',
                           width = 60,
                           height = 5,
                           state = tkinter.DISABLED) # 指定按钮为禁用状态
button4.pack()
root.mainloop()                                # 进入消息循环
```

【代码说明】代码中分别应用不同的参数实例化了四种按钮，并将其依次加入主窗口中。

【运行效果】如图 12.3 所示，在主程序窗口中显示出四种不同的按钮。

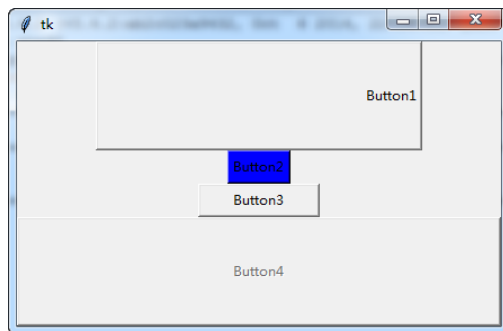


图 12.3 不同类型的按钮



12.4.2 文本框

文本框主要用来接收用户输入。

使用 `tkinter.Entry` 和 `tkinter.Text` 可以创建单行文本框和多行文本框组件。通过向其传递参数可以设置文本框的背景色、大小、状态等。`tkinter.Entry` 和 `tkinter.Text` 共有的几个控制参数及其作用如表 12.4 所示。

表 12.4 文本框的控制参数表

参 数 名	作 用
<code>background (bg)</code>	指定文本框的背景色
<code>borderwidth (bd)</code>	指定文本框边框的宽度
<code>font</code>	指定文本框中文字的字体
<code>foreground (fg)</code>	指定文本框的前景色
<code>selectbackground</code>	指定选定文本的背景色
<code>selectforeground</code>	指定选定文本的前景色
<code>show</code>	指定文本框中显示的字符，若为 “*”，表示文本框为密码框
<code>state</code>	指定文本框的状态
<code>width</code>	指定文本框的宽度

【实例 12-4】演示了在主窗口中显示创建的各种不同类型的文本框，代码如下：

```
# -*- coding:utf-8 -*-
#
import tkinter                                # 导入 tkinter 模块
root = tkinter.Tk()
entry1 = tkinter.Entry(root,                  # 生成单行文本框组件
                        show = '*',)          # 输入文本框中的字符被显示为 “*”
entry1.pack()                                 # 将文本框添加到窗口中
entry2 = tkinter.Entry(root,                  # 输入文本框中的字符被显示为 “#”
                        show = '#',           # 将文本框的宽度设置为 50
                        width = 50)
entry2.pack()
entry3 = tkinter.Entry(root,                  # 将文本框的背景色设置为红色
                        bg = 'red',            # 将文本框的前景色设置为蓝色
                        fg = 'blue')
entry3.pack()
entry4 = tkinter.Entry(root,                  # 将选中文本的背景色设置为红色
                        selectbackground = 'red', # 将选中文本的前景色设置为灰色
                        selectforeground = 'gray')
entry4.pack()
entry5 = tkinter.Entry(root,                  # 将文本框设置为禁用
                        state = tkinter.DISABLED)
entry5.pack()
edit1 = tkinter.Text(root,                   # 生成多行文本框组件
                      selectbackground = 'red', # 将选中文本的背景色设置为红色
                      selectforeground = 'gray') # 将选中文本的前景色设置为灰色
edit1.pack()
root.mainloop()                             # 进入消息循环
```

【代码说明】代码中分别应用不同的参数实例化了六种文本框，并将其依次加入主窗口中。

【运行效果】如图 12.4 所示，在主窗口中显示了不同类型的文本框和输入效果。

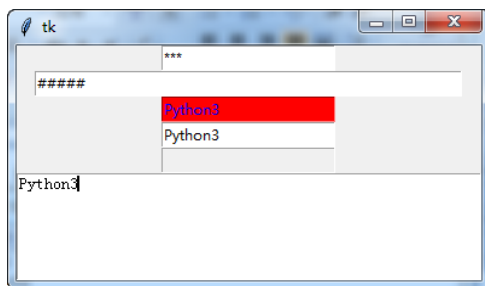


图 12.4 不同类型的文本框

12.4.3 标签

标签是提供在窗口中显示文本的组件。除显示文本以外，标签还可以显示图片。使用 `tkinter.Label` 可以创建标签组件。控制标签的参数及其作用如表 12.5 所示。

表 12.5 标签控制参数表

参 数 名	作 用
<code>anchor</code>	指定标签中文本的位置
<code>background (bg)</code>	指定标签的背景色
<code>borderwidth (bd)</code>	指定标签的边框宽度
<code>bitmap</code>	指定标签中的位图
<code>font</code>	指定标签中文本的字体
<code>foreground (fg)</code>	指定标签的前景色
<code>height</code>	指定标签的高度
<code>image</code>	指定标签中的图片
<code>justify</code>	指定标签中多行文本的对齐方式
<code>text</code>	指定标签中的文本，可以使用“\n”表示换行
<code>width</code>	指定标签的宽度

【实例 12-5】演示了在主窗口中显示创建的各种不同类型的标签组件，代码如下：

```
# -*- coding:utf-8 -*-
#
import tkinter                                # 导入tkinter 模块
root = tkinter.Tk()
label1 = tkinter.Label(root,
                        anchor = tkinter.E,      # 设置文本的位置
                        bg = 'blue',            # 设置标签背景色
                        fg = 'red',             # 设置标签前景色
                        text = 'Python',         # 设置标签中的文本
                        width = 30,             # 设置标签的宽度为 30
                        height = 5)             # 设置标签的高度为 5
label1.pack()
label2 = tkinter.Label(root,
                        text = 'Python GUI\ntkinter', # 设置标签中的文本，在字符串中使用换行符
                        justify = tkinter.LEFT,      # 设置多行文本为左对齐
                        width = 30,
                        height = 5)
label2.pack()
label3 = tkinter.Label(root,
```



```

        text = 'Python GUI\ntkinter',
        justify = tkinter.RIGHT,          # 设置多行文本为右对齐
        width = 30,
        height = 5)
label3.pack()
label4 = tkinter.Label(root,
        text = 'Python GUI\ntkinter',
        justify = tkinter.CENTER,        # 设置多行文本为局中对齐
        width = 30,
        height = 5)
label4.pack()
root.mainloop()

```

【代码说明】代码中分别应用不同的参数实例化了四种标签，并将其依次加入主窗口中。

【运行效果】如图 12.5 所示，在主窗口中显示了不同类型的标签。

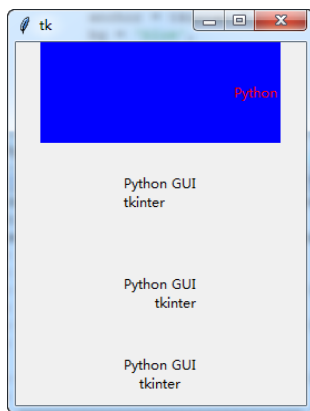


图 12.5 不同类型的标签

12.4.4 菜单

在 tkinter 中，菜单组件的添加与其他的组件有所不同。菜单要使用创建的主窗口的 config 方法添加到窗口中。

【实例 12-6】演示了添加菜单的主窗口，代码如下：

```

# -*- coding:utf-8 -*-
#
import tkinter
root = tkinter.Tk()
menu = tkinter.Menu(root)
submenu = tkinter.Menu(menu, tearoff=0)
submenu.add_command(label="Open")
submenu.add_command(label="Save")
submenu.add_command(label="Close")
menu.add_cascade(label="File", menu=submenu)
submenu = tkinter.Menu(menu, tearoff=0)
submenu.add_command(label="Copy")
submenu.add_command(label="Paste")
submenu.add_separator()
submenu.add_command(label="Cut")
menu.add_cascade(label="Edit", menu=submenu)
submenu = tkinter.Menu(menu, tearoff=0)
submenu.add_command(label="About")

# 导入 tkinter 模块
# 生成菜单
# 生成下拉菜单
# 向下拉菜单中添加 Open 命令
# 向下拉菜单中添加 Save 命令
# 向下拉菜单中添加 Close 命令
# 将下拉菜单添加到菜单中
# 生成下拉菜单
# 向下拉菜单中添加 Copy 命令
# 向下拉菜单中添加 Paste 命令
# 向下拉菜单中添加分隔符
# 向下拉菜单中添加 Cut 命令
# 将下拉菜单添加到菜单中
# 生成下拉菜单
# 向下拉菜单中添加 About 命令

```

```

menu.add_cascade(label="Help", menu=submenu)           # 将下拉菜单添加到菜单中
root.config(menu=menu)
root.mainloop()

```

【代码说明】代码中分别在主窗口加入了三个主菜单，而每个主菜单又各自具有子菜单。

【运行效果】如图 12.6 所示，在主窗口中显示了三个主菜单（File，Edit，Help），图中还显示出了 File 主菜单下的三个子菜单（Open，Save，Close）。

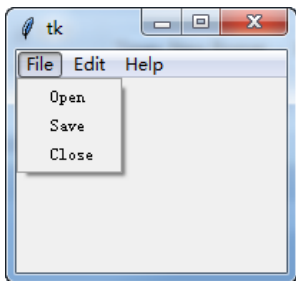


图 12.6 具有菜单的主窗口

创建弹出式菜单与上述创建菜单的过程类似，只要在单击鼠标右键后使用菜单的 `post` 方法显示菜单即可。

【实例 12-7】演示了添加弹出式菜单（快捷菜单或右键菜单）的主窗口，代码如下：

```

# -*- coding:utf-8 -*-
#
import tkinter
root = tkinter.Tk()
menu = tkinter.Menu(root, tearoff=0)                # 创建菜单
menu.add_command(label="Copy")                       # 向弹出式菜单中添加 Copy 命令
menu.add_command(label="Paste")                     # 向弹出式菜单中添加 Paste 命令
menu.add_separator()                                # 向弹出式菜单中添加分隔符
menu.add_command(label="Cut")                       # 向弹出式菜单中添加 Cut 命令
def popupmenu(event):                               # 定义右键事件处理函数
    menu.post(event.x_root, event.y_root)           # 显示菜单
root.bind("<Button-3>", popupmenu)                 # 在主窗口中绑定右键事件
root.mainloop()

```

【代码说明】代码首先建立了菜单，并向其中添加了三个子菜单，之后在主窗口中绑定了右键单击事件，单击鼠标右键时弹出快捷菜单。

【运行效果】如图 12.7 所示，在窗口中单击鼠标右键将显示弹出式菜单。

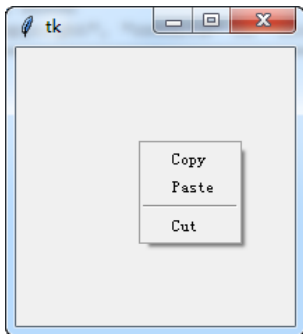


图 12.7 创建弹出式菜单



12.4.5 单选框和复选框

单选框往往用于一组互斥的选项，即一组单选框中只有一个可以被选中。而复选框则由一个复选框组件来表示两种不同的状态，即被选中表示一种状态，未被选中表示另一种状态。

使用 `tkinter.Radiobutton` 和 `tkinter.Checkbutton` 可以分别创建单选框和复选框。通过向其传递参数可以设置单选框和复选框的背景色、大小、状态等。以下是 `tkinter.Radiobutton` 和 `tkinter.Checkbutton` 共有的控制参数及其作用，如表 12.6 所示。

表 12.6 单选框和复选框控制参数表

参 数 名	作 用
<code>anchor</code>	指定文本位置
<code>background (bg)</code>	指定背景色
<code>borderwidth (bd)</code>	指定边框的宽度
<code>bitmap</code>	指定组件中的位图
<code>font</code>	指定组件中文本的字体
<code>foreground (fg)</code>	指定组件的前景色
<code>height</code>	指定组件的高度
<code>image</code>	指定组件中的图片
<code>justify</code>	指定组件中多行文本的对齐方式
<code>text</code>	指定组件中的文本，可以使用“\n”表示换行
<code>value</code>	指定组件被选中后关联变量的值
<code>variable</code>	指定组件所关联的变量
<code>width</code>	指定组件的宽度

对于单选框和复选框，`variable` 是比较关键的参数。由 `variable` 指定的变量应使用 `tkinter.IntVar` 或 `tkinter.StringVar` 生成。其中 `tkinter.IntVar` 生成一个整型变量，而 `tkinter.StringVar` 将生成一个字符串变量。

当使用 `tkinter.IntVar` 或者 `tkinter.StringVar` 生成变量后，可以使用 `set` 方法设置变量的初始值。如果该初始值与组件的 `value` 所指定的值相等，则该组件处于被选中的状态。如果其他组件被选中，则变量值将被更改为该组件 `value` 所指定的值。

【实例 12-8】 演示了创建一组单选框和一个复选框 GUI 实例程序，代码如下：

```
# -*- coding:utf-8 -*-
#
import tkinter                                # 导入 tkinter 模块
root = tkinter.Tk()
r = tkinter.StringVar()                       # 使用 StringVar 生成字符串变量用于单选框组件
r.set('1')                                    # 初始化变量值
radio = tkinter.Radiobutton(root,             # 生成单选框组件
                             variable = r,    # 设置单选框关联的变量
                             value = '1',     # 设置选中单选框时其所关联的变量的值，即 r 的值
                             text = 'Radio1') # 设置单选框显示的文本
radio.pack()
radio = tkinter.Radiobutton(root,
                             variable = r,
                             value = '2',     # 当选该单选框时，r 的值为 2
                             text = 'Radio2' )
radio.pack()
```

```

radio = tkinter.Radiobutton(root,
                             variable = r,
                             value = '3',
                             text = 'Radio3' )
radio.pack()
radio = tkinter.Radiobutton(root,
                             variable = r,
                             value = '4',
                             text = 'Radio4' )
radio.pack()
c = tkinter.IntVar()
c.set(1)
check = tkinter.Checkbutton(root,
                             text = 'Checkbutton',
                             variable = c,
                             onvalue = 1,
                             offvalue = 2)
check.pack()
root.mainloop()
print(r.get())
print(c.get())

```

当选中该单选框时, r 的值为 3

当选中该单选框时, r 的值为 4

使用 IntVar 生成整型变量用于复选框

设置复选框的文本

设置复选框关联的变量

当选中复选框时, c 的值为 1

当未选中复选框时, c 的值为 2

输出 r 的值

输出 c 的值

【代码说明】代码中首先创建了一个包含四项的单选框和一个复选框, 并把它们加入到主窗口中。

【运行效果】如图 12.8 所示, 在窗口中显示了单选框和复选框。

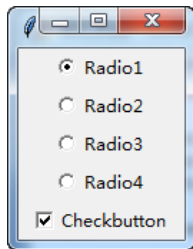


图 12.8 创建单选框和复选框



注意 在创建单选框和复选框前应创建其相关联的变量。

对于单选框组件和复选框组件, 还有一个比较特殊的控制参数 `indicatoron`, 当向其传递值 0 时, 组件将被绘制成按钮的形式, 被选中的组件处于按下状态。

【实例 12-9】演示了创建按钮形式的单选框和复选框 GUI 实例程序, 代码如下:

```

# -*- coding:utf-8 -*-
#
import tkinter
root = tkinter.Tk()
r = tkinter.StringVar()
r.set('1')
radio = tkinter.Radiobutton(root,
                             variable = r,
                             value = '1',
                             indicatoron = 0,
                             text = 'Radio1')
radio.pack()
radio = tkinter.Radiobutton(root,

```

导入 tkinter 模块

使用 StringVar 生成字符串变量用于单选框组件

初始化变量值

生成单选框组件

设置单选框关联的变量

设置选中单选框时其所关联的变量的值, 即 r 的值

将单选框绘制成按钮样式

设置单选框显示的文本



```
variable = r,
value = '2',
indicatoron = 0,
text = 'Radio2' )
radio.pack()
radio = tkinter.Radiobutton(root,
variable = r,
value = '3',
indicatoron = 0,
text = 'Radio3' )
radio.pack()
radio = tkinter.Radiobutton(root,
variable = r,
value = '4',
indicatoron = 0,
text = 'Radio4' )
radio.pack()
c = tkinter.IntVar()
c.set(1)
check = tkinter.Checkbutton(root,
text = 'Checkbutton',
variable = c,
indicatoron = 0,
onvalue = 1,
offvalue = 2)
check.pack()
root.mainloop()
```

【代码说明】代码中首先创建了一个包含四项按钮形式的单选框和一个按钮形式的复选框，并把它们加入到主窗口中。

【运行效果】如图 12.9 所示，在窗口中显示了按钮形式的单选框和复选框。



图 12.9 按钮样式的单选框和复选框

12.4.6 绘制图形

使用 tkinter.Canvas 创建 Canvas 绘图组件后，可以使用 Canvas 提供的方法在 Canvas 组件中绘制直线、圆弧、矩形以及图片等。Canvas 绘图组件的控制参数及其作用如表 12.7 所示。

表 12.7 绘图组件控制参数表

参 数 名	作 用
background (bg)	指定绘图组件的背景色
borderwidth (bd)	指定绘图组件的边框宽度
bitmap	指定绘图组件中的位图
foreground (fg)	指定绘图组件的前景色



续表

参 数 名	作 用
height	指定绘图组件的高度
image	指定绘图组件中的图片
width	指定绘图组件的宽度

Canvas 绘图组件的绘图方法及其功能如表 12.8 所示。

表 12.8 绘图组件方法表

方 法 名	功 能
create_arc	绘制圆弧
create_bitmap	绘制位图, 支持 XBM
create_image	绘制图片, 支持 GIF
create_line	绘制直线
create_oval	绘制椭圆
create_polygon	绘制多边形
create_rectangle	绘制矩形
create_text	绘制文字
create_window	绘制窗口
delete	删除绘制的图形

【实例 12-10】演示了使用 Canvas 绘图组件的绘图方法绘制不同图形的实例程序，代码如下：

```
# -*- coding:utf-8 -*-
#
import tkinter                                # 导入tkinter 模块
root = tkinter.Tk()
canvas = tkinter.Canvas(root,
                           width = 600,        # 指定 Canvas 组件的宽度为 600
                           height = 480,       # 指定 Canvas 组件的高度为 480
                           bg = 'white')       # 指定 Canvas 组件的背景色为白色
im = tkinter.PhotoImage(file='python.gif')    # 使用 PhotoImage 打开图片
canvas.create_image(300,50,image = im)         # 使用 create_image 将图片添加到
                                              # Canvas 组件中
canvas.create_text(302,77,
                  text = 'Use Canvas'
                  ,fill = 'gray')              # 使用 create_text 方法绘制文字
                                              # 所绘制文字的内容
canvas.create_text(300,75,
                  text = 'Use Canvas',
                  fill = 'blue')              # 所绘制文字的颜色为灰色
canvas.create_polygon(290,114,316,114,
                     330,130,310,146,284,146,270,130) # 使用 create_polygon 方法绘制六边形
canvas.create_oval(280,120,320,140,
                  fill = 'white')              # 使用 create_oval 方法绘制椭圆
                                              # 设置椭圆用白色填充
canvas.create_line(250,130,350,130)           # 使用 create_line 绘制直线
canvas.create_line(300,100,300,160)
canvas.create_rectangle(90,190,510,410,
                       width=5)               # 使用 create_rectangle 绘制一个矩形
                                              # 设置矩形线宽为 5 像素
canvas.create_arc(100, 200, 500, 400,
                  start=0, extent=240,        # 使用 create_arc 绘制圆弧
                  )                          # 设置圆弧的起止角度
```



```

        fill="pink")                                # 设置圆弧填充颜色为粉色
canvas.create_arc(103,203,500,400,
                  start=241, extent=118,
                  fill="red")
canvas.pack()                                       # 将 Canvas 添加到主窗口
root.mainloop()

```

【代码说明】代码中首先创建了一个 Canvas，并在其上绘制了文字、图形、六边形、椭圆、直线、矩形和圆弧等图形，之后把 Canvas 实例添加到主窗口中。

【运行效果】如图 12.10 所示，在窗口中显示了绘制的图形效果。

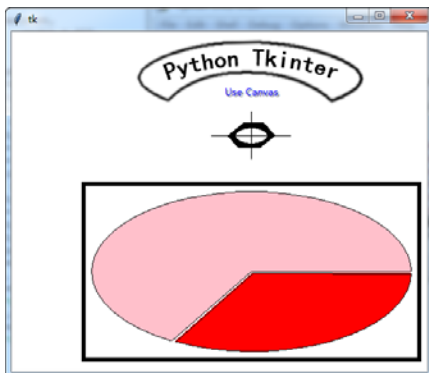


图 12.10 使用 Canvas 绘制图形

12.5 响应操作事件



tkinter 中的事件是指在各个组件上发生的各种鼠标和键盘事件。对于按钮组件、菜单组件等可以在创建组件时通过 command 参数指定其事件的处理函数。除组件所触发的事件外，在创建右键弹出菜单时，还需处理右击事件。类似的事件可以归结为鼠标事件、键盘事件和窗口事件。

12.5.1 事件基础

鼠标事件主要指鼠标按键的按下、释放，鼠标滚轮的滚动，鼠标指针移进、移出组件等所触发的事件。键盘事件主要指键的按下、释放等所触发的事件。窗口事件是指改变窗口大小、组件状态等变化所触发的事件。

对于鼠标事件、键盘事件和窗口事件，可以采用事件绑定的方法确定消息的处理方式。事件绑定可以使用组件的 bind 方法进行，或者使用 bind_class() 方法进行类绑定，分别调用函数或者类来响应事件。bind_all() 方法也可以用来绑定事件，bind_all() 方法将所有组件事件绑定到事件响应函数上。这三种方法的原型如下：

```

bind(sequence, func, add)
bind_class(className, sequence, func, add)
bind_all(sequence, func, add)

```

各参数的含义如下。

- sequence: 所绑定的事件；
- func: 所绑定的事件处理函数；
- add: 可选参数，为空字符或者“+”；
- className: 所绑定的类；

- sequence 表示所绑定的事件，必须为以“<>”包围的字符串。

鼠标事件可以使用的表示方式及其意义如表 12.9 所示。

表 12.9 鼠标事件及其意义

鼠标事件	意 义
<Button-1>	表示鼠标左键按下，而<Button-2>表示中键，<Button-3>表示右键
<ButtonPress-1>	表示鼠标左键按下，与<Button-1>相同
<ButtonRelease-1>	表示鼠标左键释放
<B1-Motion>	表示按住鼠标左键移动
<Double-Button-1>	表示双击鼠标左键
<Enter>	表示鼠标指针进入某一组件区域
<Leave>	表示鼠标指针离开某一组件区域
<MouseWheel>	表示鼠标滚轮动作

在表 12.9 所示的鼠标事件中，数字均可替换成 2 或 3。其中 2 表示鼠标中键，3 表示鼠标右键。例如<B3-Motion>表示按住鼠标右键移动，<Double-Button-2>表示双击鼠标中键等。

键盘事件可以使用的表示方式及其意义如表 12.10 所示。

表 12.10 键盘事件及其意义

键盘事件	意 义
<KeyPress-A>	表示按下 A 键，可用其他字母键代替
<Alt-KeyPress-A>	表示同时按下 Alt 和 A 键
<Control-KeyPress-A>	表示同时按下 Control 和 A 键
<Shift-KeyPress-A>	表示同时按下 Shift 和 A 键
<Double-KeyPress-A>	表示快速地按下两下 A 键
<Lock-KeyPress-A>	表示 Caps Lock 打开后按下 A 键

如表 12.10 所示的键盘事件还可以使用 Alt、Control 和 Shift 组合。例如，<Alt-Control-Shift-KeyPress-B>表示同时按下 Alt、Control、Shift 和 B 键。其中，KeyPress 可以用 KeyRelease 替换，表示当按键释放时触发事件。需要注意的是，输入的字母要区分大小写，如果使用<KeyPress-A>，则只有按下 Shift 或者 Caps Lock 打开时才触发事件。

窗口事件的几种表示方式及其意义如表 12.11 所示。


表 12.11 窗口事件及其意义

窗口事件	意 义
Activate	当组件由不可用转为可用时触发
Configure	当组件大小改变时触发
Deactivate	当组件由可用转为不可用时触发
Destroy	当组件被销毁时触发
Expose	当组件从被遮挡状态中暴露出来时触发
FocusIn	当组件获得焦点时触发
FocusOut	当组件失去焦点时触发
Map	当组件由隐藏状态变为显示状态时触发
Property	当窗体的属性被删除或改变时触发



续表

窗口事件	意 义
Unmap	当组件由显示状态变为隐藏状态时触发
Visibility	当组件变为可视状态时触发

 **注意** 要注意绑定事件到函数与绑定事件到类的区别。

12.5.2 响应事件

窗口中的事件被绑定到函数后，当该事件被触发后将调用所绑定的函数进行处理。事件触发后，系统将向该函数传递一个 event 对象的参数。因此被绑定的响应事件的函数应该定义成如下所示的形式：

```
def function(event):  
    <语句>
```

其中，event 对象具有的属性及其意义如表 12.12 所示。

表 12.12 event对象的属性及其意义

属性	意 义
char	按键字符，仅对键盘事件有效
keycode	按键名，仅对键盘事件有效
keysym	按键编码，仅对键盘事件有效
num	鼠标按键，仅对鼠标事件有效
type	所触发的事件类型
widget	引起事件的组件
width, height	组件改变后的大小，仅对 Configure 有效
x, y	鼠标当前位置，相对于窗口
x_root, y_root	鼠标当前位置，相对于整个屏幕

【实例 12-11】演示了使用事件处理创建一个简单的绘图的实例程序，代码如下：

```
# -*- coding:utf-8 -*-  
#  
import tkinter  
class MyButton:  
    def __init__(self,root,canvas,label,type):  
        self.root = root  
        self.canvas = canvas  
        self.label = label  
        if type == 0:  
            button = tkinter.Button(root,text = 'DrawLine',  
                                    command = self.DrawLine)  
        elif type == 1:  
            button = tkinter.Button(root,text = 'DrawArc',  
                                    command = self.DrawArc)  
        elif type == 2:  
            button = tkinter.Button(root,text = 'DrawRec',  
                                    command = self.DrawRec)  
        else :  
            button = tkinter.Button(root,text = 'DrawOval',  
                                    command = self.DrawOval)
```

```

        button.pack(side = 'left')
    def DrawLine(self):                                # DrawLine 按钮事件处理函数
        self.label.text.set('Draw Line')
        self.canvas.SetStatus(0)
    def DrawArc(self):                                  # DrawArc 按钮事件处理函数
        self.label.text.set('Draw Arc')
        self.canvas.SetStatus(1)
    def DrawRec(self):                                  # DrawRec 按钮事件处理函数
        self.label.text.set('Draw Rectangle')
        self.canvas.SetStatus(2)
    def DrawOval(self):                                 # DrawOval 按钮事件处理函数
        self.label.text.set('Draw Oval')
        self.canvas.SetStatus(3)
class MyCanvas:                                       # 定义 Canvas 类
    def __init__(self,root):
        self.status = 0                                # 保存引用值
        self.draw = 0
        self.root = root
        self.canvas = tkinter.Canvas(root,bg = 'white', # 生成 Canvas 组件
            width = 600,
            height = 480)
        self.canvas.pack()
        self.canvas.bind('<ButtonRelease-1>',self.Draw) # 绑定事件到左键
        self.canvas.bind('<Button-2>',self.Exit)        # 绑定事件到中键
        self.canvas.bind('<Button-3>',self.Del)         # 绑定事件到右键
        self.canvas.bind_all('<Delete>',self.Del)       # 绑定事件到 Delete 键
        self.canvas.bind_all('<KeyPress-d>',self.Del)   # 绑定事件到 d 键
        self.canvas.bind_all('<KeyPress-e>',self.Exit)  # 绑定事件到 e 键
    def Draw(self,event):                              # 绘图事件处理函数
        if self.draw == 0:                             # 判断是否绘图
            self.x = event.x
            self.y = event.y
            self.draw = 1
        else:                                           # 根据 self.status 绘制不同的图形
            if self.status == 0:
                self.canvas.create_line(self.x,self.y,
                    event.x,event.y)
                self.draw = 0
            elif self.status == 1:
                self.canvas.create_arc(self.x,self.y,
                    event.x,event.y)
                self.draw = 0
            elif self.status == 2:
                self.canvas.create_rectangle(self.x,self.y,
                    event.x,event.y)
                self.draw = 0
            else:
                self.canvas.create_oval(self.x,self.y,
                    event.x,event.y)
                self.draw = 0
    def Del(self,event):                                # 按下右键或 d 键则删除图形
        items = self.canvas.find_all()
        for item in items:
            self.canvas.delete(item)
    def Exit(self,event):                               # 按下中键或 e 键则退出
        self.root.quit()
    def SetStatus(self,status):                         # 设置绘制的图形
        self.status = status
class MyLabel:                                       # 定义标签类
    def __init__(self,root):                          # 类初始化

```



```

self.root = root                                # 保存引用
self.canvas = canvas
self.text = tkinter.StringVar()                 # 生成标签引用变量
self.text.set('Draw Line')
self.label = tkinter.Label(root, textvariable = self.text, # 生成标签
                             fg = 'red', width = 50)
self.label.pack(side = 'left')

root = tkinter.Tk()                             # 生成主窗口
canvas = MyCanvas(root)                         # 生成绘图组件
label = MyLabel(root)                           # 生成标签
MyButton(root, canvas, label, 0)                # 生成按钮
MyButton(root, canvas, label, 1)
MyButton(root, canvas, label, 2)
MyButton(root, canvas, label, 3)
root.mainloop()                                # 进入消息循环

```

【代码说明】代码中利用了类的继承，分别自定义了 `MyButton`、`MyCanvas`、`MyLabel` 三个类，各自继承了 `tkinter` 中的组件 `Button`、`Canvas`、`Label`，并添加了新的方法。随后将它们添加到主窗口中。

【运行效果】如图 12.11 所示，在窗口中单击鼠标左键，然后移动到另一位置，再单击左键将绘制图形。可以单击按钮选择要绘制的图形。单击右键或者按键盘上的 `D` 键将删除多余的图形。单击鼠标中键或者按键盘上的 `E` 键将关闭窗口。

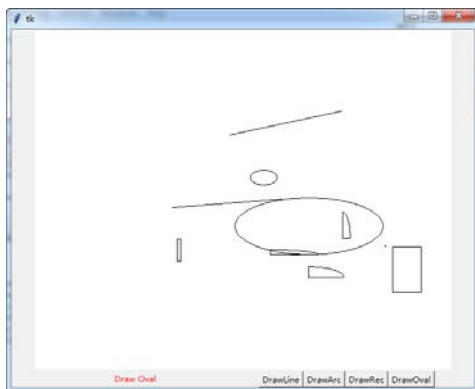


图 12.11 简单的绘图程序

12.6 对话框

在 `tkinter` 中提供了标准的对话框，在程序中可以直接使用这些标准对话框与用户交互。如果 `tkinter` 提供的对话框不能满足要求，还可以使用 `Toplevel` 来创建对话框。

12.6.1 标准对话框

标准对话框包含简单的消息框和用户输入对话框。其中，信息框以窗口的形式向用户输出信息，也可以获取用户所单击的按钮。输入对话框要求用户输入字符串、整型或者浮点型的值。

1. 消息框

`tkinter.messagebox` 模块提供了几种简单的消息框（在 `Python 2.x` 中由 `tkMessageBox` 模块提供）。使用 `tkinter.messagebox` 模块中的 `askokcancel`、`askquestion`、`askyesno`、`showerror`、`showinfo` 和 `showwarning` 可以创建简单的消息框。使用这些函数时只需向其传递 `title` 和 `message` 参数。

【实例 12-12】 演示了使用 tkMessageBox 创建简单的消息框实例程序，代码如下：

```
# -*- coding:utf-8 -*-
#
import tkinter                                # 导入 tkinter 模块
import tkinter.messagebox                    # 导入 tkMessageBox 模块
def cmd():                                    # 按钮消息处理函数
    global n                                  # 使用全局变量 n
    global buttontext                        # 使用全局变量 buttontext
    n = n + 1
    if n == 1:                               # 判断 n 的值，显示不同的消息框
        tkinter.messagebox.askokcancel('Python tkinter', 'askokcancel')
        # 使用 askokcancel 函数
        buttontext.set('skquestion')         # 更改按钮上的文字
    elif n == 2:
        tkinter.messagebox.askquestion('Python tkinter', 'skquestion')
        # 使用 askquestion 函数
        buttontext.set('askyesno')
    elif n == 3:
        tkinter.messagebox.askyesno('Python tkinter', 'askyesno')
        # 使用 askyesno 函数
        buttontext.set('showerror')
    elif n == 4:
        tkinter.messagebox.showerror('Python tkinter', 'showerror')
        # 使用 showerror 函数
        buttontext.set('showinfo')
    elif n == 5:
        tkinter.messagebox.showinfo('Python tkinter', 'showinfo')
        # 使用 showinfo 函数
        buttontext.set('showwarning')
    else :
        n = 0                                # 将 n 赋值为 0 重新开始循环
        tkinter.messagebox.showwarning('Python tkinter', 'showwarning')
        # 使用 showwarning 函数
        buttontext.set('askokcancel')
n = 0                                         # 为 n 赋初始值
root = tkinter.Tk()
buttontext = tkinter.StringVar()            # 生成关联按钮文字的变量
buttontext.set('askokcancel')               # 设置 buttontext 值
button = tkinter.Button(root,               # 生成按钮
    textvariable = buttontext,              # 设置关联变量
    command = cmd)                          # 设置事件处理函数
button.pack()
root.mainloop()                             # 进入消息循环
```

【代码说明】 代码中首先创建了按钮消息处理函数，然后将其绑定到按钮上，并加入主窗口。

【运行效果】 如图 12.12 所示，运行主程序后界面。单击主窗口中的按钮将依次创建如图 12.13 所示的消息框。

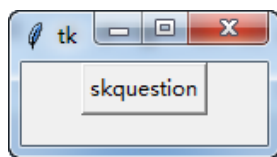


图 12.12 一个按钮的主程序

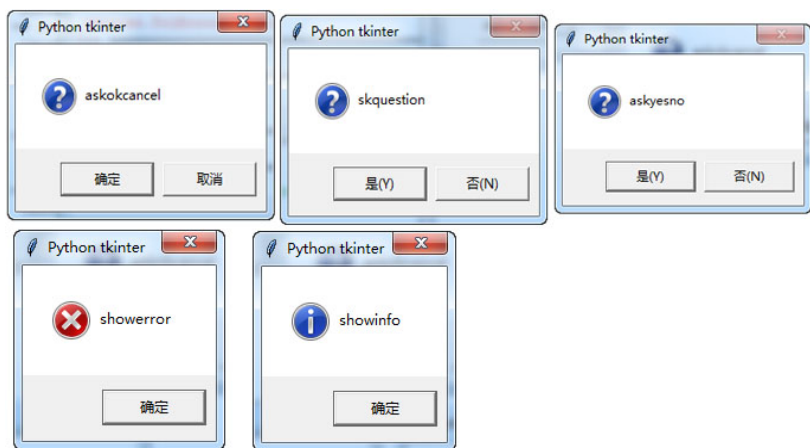


图 12.13 依次弹出各种类型的对话框

除了上述 5 个标准的消息框以外，还可以使用 `tkinter.messagebox._show` 函数创建其他类型的消息框。`tkinter.messagebox._show` 函数有以下控制参数。

- **default:** 指定消息框的按钮；
- **icon:** 指定消息框的图标；
- **message:** 指定消息框所显示的消息；
- **parent:** 指定消息框的父组件；
- **title:** 指定消息框的标题；
- **type:** 指定消息框的类型。

2. 使用标准对话框

使用 `tkinter.simpledialog` 模块、`tkinter.filedialog` 模块、`tkinter.colorchooser` 模块（在 Python 2.x 中的 `tkSimpleDialog` 模块、`tkFileDialog` 模块和 `tkColorChooser` 模块）可以创建标准的对话框。其中 `tkinter.simpledialog` 模块可以创建标准的输入对话框。`tkinter.filedialog` 模块可以创建文件打开和保存文件对话框。`tkinter.colorchooser` 模块可以创建颜色选择对话框。

`tkinter.simpledialog` 模块可以创建三种类型的对话框：输入字符串、输入整数和输入浮点型的对话框。对应的函数分别为 `askstring`、`askinteger` 和 `askfloat` 函数，其具有以下几个相同的可选参数。

- **title:** 指定对话框标题；
- **prompt:** 指定对话框中显示的文字；
- **initialvalue:** 指定输入框的初始值。

使用 `tkinter.simpledialog` 模块中的函数创建对话框后，将返回对话框中文本框的值。

【实例 12-13】 演示了具有三种简单的输入对话框的实例程序，代码如下：

```
# -*- coding:utf-8 -*-
#
import tkinter                                     # 导入 tkinter 模块
import tkinter.simpledialog                         # 导入 tkSimpleDialog 模块
def InStr():                                       # 按钮事件处理函数
    r = tkinter.simpledialog.askstring('Python tkinter', # 创建字符串输入对话框
        'Input String',                                # 指定提示字符
        initialvalue='tkinter')                      # 指定初始化文本
    print(r)                                         # 输出返回值
```

```

def InInt():                                     # 按钮事件处理函数
    r = tkinter.simpledialog.askinteger('Python tkinter','Input Integer')
                                                # 创建整数输入对话框

    print(r)

def InFlo():                                     # 按钮事件处理函数
    r = tkinter.simpledialog.askfloat('Python tkinter','Input Float')
                                                # 创建浮点数输入对话框

    print(r)

root = tkinter.Tk()
button1 = tkinter.Button(root,text = 'Input String',      # 创建按钮
                           command = InStr)              # 指定按钮事件处理函数
button1.pack(side='left')
button2 = tkinter.Button(root,text = 'Input Integer',
                           command = InInt)               # 指定按钮事件处理函数
button2.pack(side='left')
button3 = tkinter.Button(root,text = 'Input Float',
                           command = InFlo)              # 指定按钮事件处理函数
button3.pack(side='left')
root.mainloop()                                         # 进入消息循环

```

【代码说明】代码中首先定义了用于创建不同类型对话框的消息处理函数，然后将其绑定到相应的按钮上，并加入主窗口。

【运行效果】如图 12.14 所示，运行主程序后的界面。单击主窗口中的三个按钮将分别弹出如图 12.15 所示的三种不同类型的对话框。

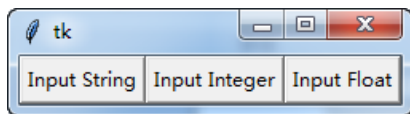


图 12.14 弹出不同对话框的程序主窗口

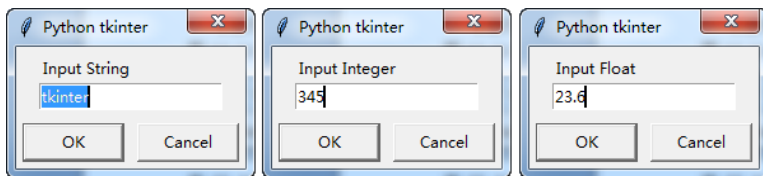


图 12.15 三种不同类型的对话框

`tkinter.filedialog` 模块中的 `askopenfilename` 函数可以创建标准的打开文件对话框。`asksaveasfilename` 可以创建标准的保存文件对话框，其具有以下几个相同的可选参数。

- `filetypes`: 指定文件类型；
- `initialdir`: 指定默认目录；
- `initialfile`: 指定默认文件；
- `title`: 指定对话框标题。

使用 `tkFileDialog` 模块中的函数创建对话框后，将返回文件的完整路径。

【实例 12-14】演示了一个创建文件打开和保存对话框的实例程序，代码如下：

```

# -*- coding:utf-8 -*-
#
import tkinter                                # 导入 tkinter 模块
import tkinter.filedialog                    # 导入 tkFileDialog 模块
def FileOpen():                              # 按钮事件处理函数
    r = tkinter.filedialog.askopenfilename(title = 'Python tkinter',

```



```

# 创建打开文件对话框
filetypes=[('Python', '*.py *.pyw'),('All files', '*')] )
# 指定文件类型为 Python 程序
# 输出返回值
# 按钮事件处理函数
def FileSave():
    r = tkinter.filedialog.asksaveasfilename(title = 'Python tkinter',
# 创建保存文件对话框
initialdir=r'E:\Python\code',
# 指定初始化目录
initialfile = 'test.py')
# 指定初始化文件
print(r)
root = tkinter.Tk()
button1 = tkinter.Button(root,text = 'File Open',# 创建按钮
command = FileOpen)
# 指定按钮事件处理函数
button1.pack(side='left')
button2 = tkinter.Button(root,text = 'File Save',
command = FileSave)
# 指定按钮事件处理函数
button2.pack(side='left')
root.mainloop()
# 进入消息循环

```

【代码说明】代码中首先定义了用于创建打开文件和保存文件的对话框的消息处理函数，然后将其绑定到相应的按钮上，并加入主窗口。

【运行效果】图 12.16 展示了运行主程序后的界面。单击“File Open”按钮，将创建如图 12.17 所示的打开文件对话框。单击“File Save”按钮，将创建如图 12.18 所示的保存文件对话框。

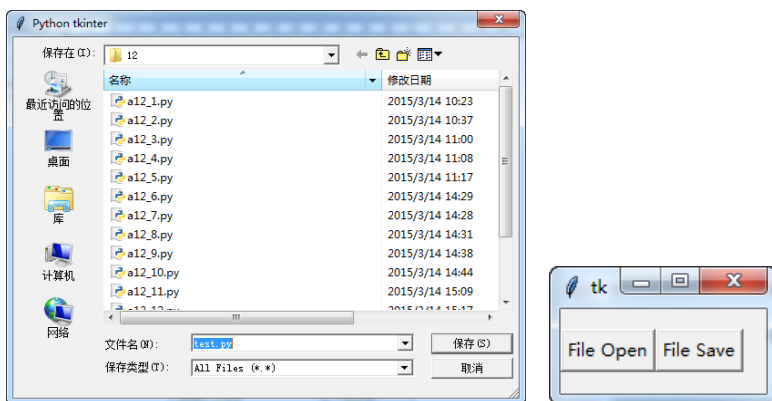


图 12.16 文件对话框程序窗口

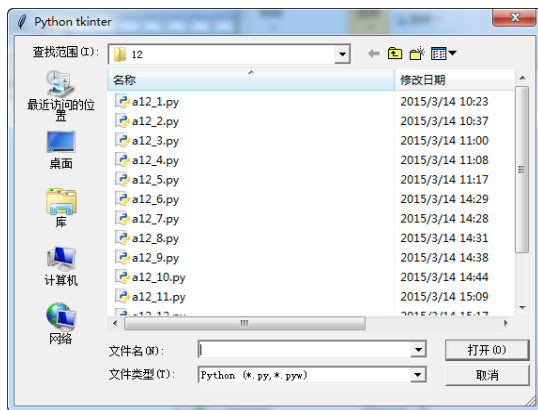


图 12.17 打开文件对话框

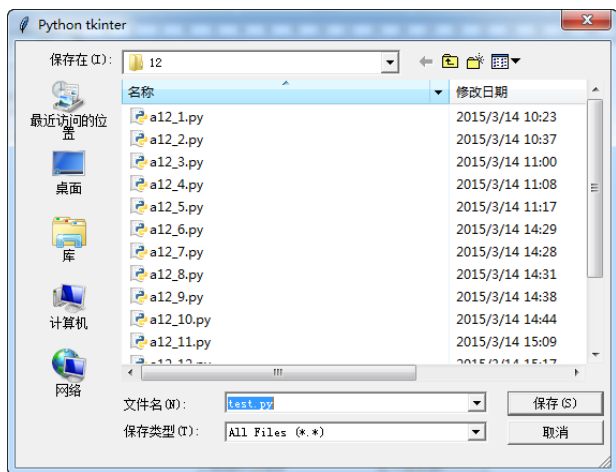


图 12.18 保存文件对话框

tkinter.colorchooser 模块中的 askcolor 函数可以创建标准的颜色选择对话框,其具有以下几个可选参数。

- initialcolor: 指定初始化颜色;
- title: 指定对话框标题。

使用 tkinter.colorchooser 模块中的函数创建对话框后,将返回颜色的 RGB 值以及可以在 Python tkinter 中使用的颜色字符值。

【实例 12-15】演示了一个创建了颜色选择对话框的实例程序,代码如下:

```
# -*- coding:utf-8 -*-
#
import tkinter                                # 导入tkinter 模块
import tkinter.colorchooser                  # 导入tkColorChooser 模块
def ChooseColor():                           # 按钮事件处理函数
    r = tkinter.colorchooser.askcolor(title = 'Python tkinter')
                                           # 创建颜色选择对话框
    print(r)                                # 输出返回值
root = tkinter.Tk()
button = tkinter.Button(root,text = 'Choose Color', # 创建按钮
                        command = ChooseColor)      # 指定按钮事件处理函数
button.pack()
root.mainloop()                             # 进入消息循环
```

【代码说明】代码中首先定义了用于创建颜色选择器的消息处理函数,然后将其绑定到相应的按钮上,并加入主窗口。

【运行效果】图 12.19 展示了运行主程序后的界面。单击“Choose Color”命令将创建如图 12.20 所示的颜色选择对话框。

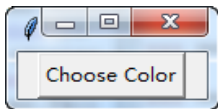


图 12.19 创建颜色选择对话框程序窗口

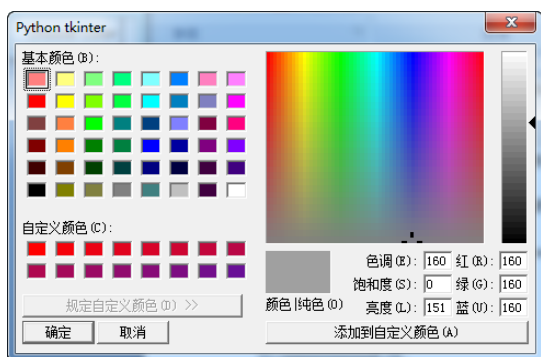


图 12.20 颜色选择对话框

12.6.2 自定义对话框

从上面介绍的内容可看出，tkinter 提供了简单的对话框，可以方便地在程序中使用，以打开标准的对话框。

如果 tkinter 所提供的对话框不能满足要求，则可以使用 Toplevel 组件来创建自定义的对话框。在程序中可以向 Toplevel 组件添加其他组件，并且定义事件响应函数或者类等。

在使用 tkinter 创建对话框的时候，如果对话框中也需要进行事件处理，最好以类的形式来定义对话框，否则只能大量使用全局变量来处理参数，导致程序维护和调试困难。对于代码较多的 tkinter GUI Python 程序，整个程序也应该使用类的方式来组织。

【实例 12-16】演示了一个使用 Toplevel 组件创建一个简单的对话框的实例程序，代码如下：

```
# -*- coding:utf-8 -*-
#
import tkinter
import tkinter.messagebox
class MyDialog:
    def __init__(self, root):
        self.top = tkinter.Toplevel(root)
        label = tkinter.Label(self.top, text='Please Input')
        label.pack()
        self.entry = tkinter.Entry(self.top)
        self.entry.pack()
        self.entry.focus()
        button = tkinter.Button(self.top, text='Ok',
                                command=self.Ok)
        button.pack()
    def Ok(self):
        self.input = self.entry.get()
        self.top.destroy()
    def get(self):
        return self.input
class MyButton():
    def __init__(self, root, type):
        self.root = root
        if type == 0:
            self.button = tkinter.Button(root,
                                          text='Create',
                                          command = self.Create)
            # 导入tkinter 模块
            # 导入 tkMessageBox 模块
            # 定义对话框类
            # 对话框初始化
            # 生成 Toplevel 组件
            # 生成标签组件
            # 生成文本框组件
            # 让文本框获得焦点
            # 生成按钮
            # 设置按钮事件处理函数
            # 定义按钮事件处理函数
            # 获取文本框中的内容，保存为 input
            # 销毁对话框
            # 返回在文本框中输入的内容
            # 定义按钮类
            # 按钮初始化
            # 保存父窗口引用
            # 根据类型创建不同按钮
            # 设置 Create 按钮事件处理函数
```

```

else:
    self.button = tkinter.Button(root,
                                text='Quit',
                                command = self.Quit)    # 设置 Quit 按钮的事件处理函数
    self.button.pack()
def Create(self):                                     # Create 按钮的事件处理函数
    d = MyDialog(self.root)                           # 生成对话框
    self.button.wait_window(d.top)                    # 等待对话框结束
    tkinter.messagebox.showinfo('Python','You input:\n' + d.get())
                                                    # 获取对话框中的输入值，并输出
def Quit(self):                                       # Quit 按钮的事件处理函数
    self.root.quit()                                  # 退出主窗口
root = tkinter.Tk()                                  # 生成主窗口
MyButton(root,0)                                     # 生成 Create 按钮
MyButton(root,1)                                     # 生成 Quit 按钮
root.mainloop()                                      # 进入消息循环

```

【代码说明】代码中首先自定义了 MyDialog 和 MyButton 类，实例化后加入主窗口。

【运行效果】运行程序后将创建如图 12.21 所示的窗口，单击“Create”按钮后将创建如图 12.22 所示的对话框，在其中的文本框中输入一些文字，单击“OK”按钮后，将弹出如图 12.23 所示的信息框。在图 12.21 所示的窗口中单击“Quit”按钮，则退出窗口。

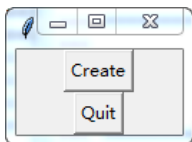


图 12.21 主程序窗口

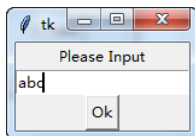


图 12.22 自定义对话框



图 12.23 信息框

12.7 小结

本章介绍了可在不同操作系统中使用的 tkinter 模块，这是 Python 内置的 GUI 设计模块，可直接使用。首先介绍了 tkinter 模块的一些基本概念，接着介绍了 tkinter 模块中常用组件的创建方法及其常用属性，然后介绍了 tkinter 模块事件处理的方法，最后介绍了使用 tkinter 标准对话框及创建自定义对话框的方法。通过学习本章，你应掌握使用 tkinter 库来进行 GUI 编程的基本知识，能运用它编写一些基本的 GUI 界面的 Python 应用程序。

12.8 本章习题

一、简答题

1. 什么是 GUI 程序？



2. 用 tkinter 创建 GUI 程序的基本步骤是什么？
3. 什么是 GUI 组件？其作用是什么？
4. tkinter 中布局组件有哪几种？其布局的方法是怎样的？

二、实验题

1. 使用 tkinter 库和 Python 标准库中的 math 模块编程实现常用数学函数计算器。主要功能包括：acos、asin、atan、cos、sin、exp、log、pow、sqrt、tan 函数的计算功能，并用按钮来实现数字输入和函数功能的调用操作，以文本框显示计算结果。

2. 在上题程序的基础上，为计算器添加时间显示模块、菜单、快捷菜单以实现计算结果的复制与粘贴及退出程序。

第 13 章 正则表达式

最初的正则表达式出现于理论计算机科学的自动控制理论和形式化语言理论中。在这些领域中有对计算（自动控制）的模型和对形式化语言描述与分类的研究。

程序员所用的正则表达式是指用某种模式去匹配一类具有共同特征的字符串。正则表达式主要用于处理文本，它能使处理文本简单起来，尤其对于复杂的查找、替换这样的工作，使用正则表达式会完成得非常快。流行的文本编辑器（如 Emacs、Vim 等）大都支持正则表达式。

本章主要介绍正则表达式的基本概念和 Python 标准库中 `re` 库的使用，内容包括：

- 正则表达式的基本元字符；
- 常用正则表达式分析；
- 使用 `re` 模块处理正则表达式；
- 编译生成正则表达式对象；
- 用正则表达式对象提速；
- 正则表达式中的分组；
- 匹配和搜索的结果对象：Match 对象；
- 使用正则表达式处理文件。

13.1 正则表达式基础



正则表达式又称正规表示式、正规表示法、正规表达式、规则表达式、常规表示法（Regular Expression，在代码中常简写为 `regex`、`regexp` 或 `re`），是计算机科学的一个概念。本节详细介绍了正则表达式的概念和应用。

13.1.1 正则表达式概述

正则表达式使用单个字符串来描述、匹配一系列符合某个句法规则的字符串。在很多文本编辑器里，正则表达式通常被用来检索、替换那些符合某个模式的文本。

许多程序设计语言都支持利用正则表达式进行字符串操作。例如，在 Perl 中就内建了一个功能强大的正则表达式引擎。正则表达式这个概念最初是由 UNIX 中的工具软件（例如 `sed` 和 `grep`）普及开的。正则表达式通常缩写成“`regex`”，单数有 `regexp`、`regex`，复数有 `regexps`、`regexes`、`regexen`。

正则表达式主要用于快速地搜索、替换或验证具有特殊形式或格式的文本，可以应用于文本编辑、查找，也可以应用于 web 数据处理与分析等领域。

13.1.2 正则表达式基本元字符

元字符是正则表达式中具有特定含义的字符，在正则表达式中，可以在字符串中使用元字符来匹配字符串各种可能的情况。常用的元字符及其含义如表 13.1 所示。



表 13.1 正则表达式元字符表

元字符	含 义
.	匹配除换行符以外的任何单个字符，如“r.d”会匹配“red”“r d”等，但不会匹配“read”
*	匹配位于*之前的 0 个或多个字符，如“r*ed”会匹配“ed”“rred”“rrred”“red”等
+	匹配位于+之前的一个或多个字符，如“r+ed”会匹配“rred”“rrred”，但不会匹配“ed”
	匹配位于 之前或者之后的字符，如“red blue”会匹配“red”“blue”
^	匹配行首
\$	匹配行尾
?	匹配位于? 之前的 0 个或一个字符，如“r?ed”会匹配“ed”“red”等，但不会匹配“rrred”
\	表示位于\之后的为转义字符
[]	匹配位于[]中的任何一个字符，如 r[ae]d, 会匹配“rad”和“red”等
()	将位于()内的内容当作一个整体
{ }	按{}中的次数进行匹配
^	匹配输入字符串的开始位置。如果设置了 RegExp 对象的 Multiline 属性，^也匹配“\n”或“\r”之后的位置
x y	匹配 x 或 y。例如，“z food”能匹配“z”或“food”。“(z f)ood”则匹配“zood”或“food”。
[xyz]	字符集合（character class）。匹配所包含的任意一个字符。例如，“[abc]”可以匹配“plain”中的“a”。特殊字符仅有反斜线\保持特殊含义，用于转义字符。其他特殊字符如星号、加号、各种括号等均作为普通字符。脱字符^如果出现在首位，则表示负值字符集合；如果出现在字符串中间，就仅作为普通字符。连字符-如果出现在字符串中间，则表示字符范围描述；如果出现在首位则仅作为普通字符
[^xyz]	排除型（negate）字符集合。匹配未列出的任意字符。例如，“[^abc]”可以匹配“plain”中的“plin”
[a-z]	字符范围。匹配指定范围内的任意字符。例如，“[a-z]”可以匹配“a”到“z”范围内的任意小写字母字符
[^a-z]	排除型的字符范围。匹配任何不在指定范围内的任意字符。例如，“[^a-z]”可以匹配任何不在“a”到“z”范围内的任意字符
\b	匹配一个单词边界，也就是指单词和空格间的位置。例如，“er\b”可以匹配“never”中的“er”，但不能匹配“verb”中的“er”
\B	匹配非单词边界。“erB”能匹配“verb”中的“er”，但不能匹配“never”中的“er”
\cx	匹配由 x 指明的控制字符。例如，\cM 匹配一个 Control-M 或回车符。x 的值必须为 A~Z 或 a~z 之一，否则，将 c 视为一个原义的“c”字符
\d	匹配一个数字字符。等价于[0-9]
\D	匹配一个非数字字符。等价于[^0-9]
\f	匹配一个换页符。等价于\x0c 和\cL
\n	匹配一个换行符。等价于\x0a 和\cJ
\r	匹配一个回车符。等价于\x0d 和\cM
\s	匹配任何空白字符，包括空格、制表符、换页符等。等价于[\f\n\r\t\v]
\S	匹配任何非空白字符。等价于[^ \f\n\r\t\v]
\t	匹配一个制表符。等价于\x09 和\cI
\v	匹配一个垂直制表符。等价于\x0b 和\cK
\w	匹配包括下划线的任何单词字符。等价于 “[A-Za-z0-9_]”
\W	匹配任何非单词字符。等价于 “[^A-Za-z0-9_]”

元字符还可以配合起来使用。“.”可匹配任意一个字符，如“r.*d”会匹配“rd”“red”“read”等。“+”可以匹配任意一个或者多个字符，如“r.+d”会匹配“red”“read”，但不会匹配“rd”。“?”可以匹配任意的零个或一个字符，如“r.?d”会匹配“rd”“red”，但不会匹配“read”。

“^”匹配行首，对下面的一段文字，“^red”只会匹配文中的第三个“red”，而对于“red\$”，则只会匹配文中的第二个“red”：

```
a red hat
blue and red
red and blue
```

在“[]”中还可以使用“-”来表示某一范围。例如，在“[]”中，“[a-z]”表示从“a”到“z”的所有小写字母，同样，“[A-Z]”表示从“A”到“Z”的所有大写字母，而“[0-9]”表示从“0”到“9”的数字。“[a-zA-Z0-9]”表示任意的字母或者数字。

原始字符串是为正则表达式设计的，以提高正则表达式的可读性，减少“\”在正则表达式中的数目。由于在正则表达式中也要使用以“\”开头的字符以表示某些特殊的含义。而在字符串中，转义字符也是以“\”开头的，这就导致了冲突。例如，在正则表达式中，“\b”表示匹配一个单词的开始或者结束，而在字符串中，“\b”则表示退格。如果在正则表达式中使用“\b”，则应该写成“\\b”。在`re.compile()`中使用“\b”的正确写法如下：

```
re.compile('\\ba.?')
```

如果使用原始字符串，则写法如下：

```
re.compile(r'\\ba.?')
```

如果要在正则表达式中匹配一个以“\”开头的字符串，比如“\word”，则首先要将“\”转义，以避免元字符“\w”，应将其写成“\\word”。而“\\word”作为一个包含有两个“\”字符串，又需要两个“\”将其转义，因此正确的写法如下：

```
re.compile('\\\\word')
```

如果使用原始字符串，则正确的写法如下：

```
re.compile(r'\\word')
```



注意 正则表达式的元字符有很多，只有经常使用它，才能熟练识记。

13.1.3 常用正则表达式

使用正则表达式可以简化程序设计。例如，在文本文件中包含一些联系人的手机号码，如果需要将使用联通和移动号码的人区分出来，可以使用正则表达式分别匹配。联通手机号的第3位是0、1或2，而移动的则是4~9中的某一个数字，可以据此来判断。以下正则表达式匹配的是联通的手机号：

```
13[0-2][0-9]{8}
```

其中，13表示匹配手机号的前两位。[0-2]匹配手机号的第3位，表示0~2之间的任何一个数字。[0-9]{8}组合起来匹配手机号剩下的8位。[0-9]表示0~9之间的任何一个数字，{8}表示匹配[0-9]8次，也就是匹配任意一个8位数。以下正则表达式匹配的是移动的手机号：

```
13[4-9][0-9]{8}
```

其与匹配联通的手机号唯一不同的地方是第3位是用[4-9]。[4-9]表示4~9之间的任意一个数字。



提示 以上只是处理以13开头的手机号码，随着手机号码的不断推出，现在还有以14、15、18开头的手机号。这里为了简化正则表达式，使初学者能看懂，暂不处理其他号段。

同样，如果联系人的信息中还包含邮政编码，而又需要将其按地区来区分，也可以使用正



则表达式来处理。对邮政编码的匹配则相对简单一些，例如，北京的邮政编码前 3 位为 100。采用与匹配手机号同样的方式，只需在 100 之后匹配任意一个 3 位数：

```
100[0-9]{3}
```

使用正则表达式匹配数字十分方便，但如果匹配字符串，则需要考虑较多的情况。例如，需要找出某一文件中所有的网址，则比较复杂。以“http://www.python.org”为例，这个网址可以分成四部分，首先是“http://”，然后为“www”，再就是站名“python”，剩下的是后缀“org”。可能的网址书写完整，具有以上四个部分。而有些网址则不规范（如不包含“http://”部分）。

将“http://www”当作一个部分，其可能的情况为“http://www”或者“www”，这一部分的正则表达式匹配可以写为以下形式：

```
(http://www|www)      # 使用“()”表示其为一个整体，使用“|”表示其中任何一个满足则匹配
```

中间的站名作为一部分，这部分可能为字母、数字或者“-”，因此该部分的正则表达式匹配可以写成以下形式：

```
[a-z0-9-]*             # [a-z0-9-]表示字母、数字或者“-”，“*”表示匹配 0 个或多个前边的字符
```

剩下的后缀考虑到可能为两个或三个的字符，因此写成以下形式：

```
[a-z]{2,3}             # {2,3}表示重两次或三次，即匹配由两个或三个字母组成的字符串
```

由于其中还包含“.”，它在正则表达式中具有特殊含义，需要将其使用“\”转义。完整的正则表达式如下：

```
(http://www|www)\.[a-z0-9-]*\.[a-z]{2,3}
```

此处没有考虑诸如“com.cn”这样的形式，读者可以试着自己将其完成。

13.2 re 模块

re 模块是 Python 语言提供的处理正则表达式的标准库，在该模块中，既可以直接匹配正则表达式的基本函数，也可以通过编译正则表达式对象，并使用 re 模块方法来使用正则表达式。

13.2.1 正则匹配搜索函数

re.match()函数用于在字符串中匹配正则表达式，如果匹配成功，则返回 MatchObject 对象实例。re.search()函数用于在字符串中查找正则表达式，如果找到则返回 MatchObject 对象实例。re.findall()函数用于在字符串中查找所有符合正则表达式的字符串，并返回这些字符串的列表。如果在正则表达式中使用了组，则返回一个元组。

re.match()函数和 re.search()函数的作用基本一样。不同的是，re.match()函数只从字符串中第一个字符开始匹配。而 re.search()函数则搜索整个字符串。以上三个函数的原型如下：

```
re.match( pattern, string[, flags])
re.search( pattern, string[, flags])
re.findall( pattern, string[, flags])
```

其参数的含义相同，具体如下：

- pattern 匹配模式；
- string 要进行匹配的字符串；
- flags 可选参数，进行匹配的标志。

注意

区分 `match()` 函数和 `search()` 函数的功能，一个只能从第一个字符开始匹配，一个可以从要匹配的字符串的中间任意一个字符进行匹配。

参数 `flags` 的选项及其意义如表 13.2 所示。

表 13.2 flags 的选项及其意义

选 项	意 义
<code>re.I</code>	忽略大小写
<code>re.L</code>	根据本地设置而更改 <code>\w</code> 、 <code>\W</code> 、 <code>\b</code> 、 <code>\B</code> 、 <code>\s</code> ，以及 <code>\S</code> 的匹配内容
<code>re.M</code>	多行匹配模式
<code>re.S</code>	使 “.” 元字符也匹配换行符
<code>re.U</code>	匹配 Unicode 字符
<code>re.X</code>	忽略 <code>pattern</code> 中的空格，并且可以使用 “#” 注释

上述匹配标志可以同时使用，同时使用几个编译标志时，需要使用 “|” 对共用的编译标志进行运算。

在交互式环境下使用上述函数进行匹配和搜索的示例代码如下：

```
>>> import re                                # 导入 re 模块
>>> s = 'Life can be good'                    # 定义字符串
>>> print(re.match('can',s) )                 # 在字符串中匹配 “can”
None                                           # 输出为 None 表示未找到
>>> print(re.search('can',s))                 # 在字符串中搜索 “can”
<_sre.SRE_Match object at 0x010DAB48>         # 返回一个 Match object，表示找到
>>> print(re.match('l.*',s))                  # 匹配任何以字母 “l” 开头的字符串
None                                           # 表示未找到
>>> print(re.match('l.*',s,re.I))             # 此处设置忽略大小写
<_sre.SRE_Match object at 0x010DAC28>         # 返回一个 Match object，表示找到
>>> re.findall('[a-z]{3}',s)                   # 查找所有 3 个字母的小写字符串
['ife', 'can', 'goo']
>>> re.findall('[a-z]{1,3}',s)                 # 查找所有由 1 到 3 个字母组成的小写字符串
['ife', 'can', 'be', 'goo', 'd']
```

13.2.2 sub()与 subn()函数

`re.sub()` 函数用于替换在字符串中符合正则表达式的内容，它返回替换后的字符串。`re.subn()` 函数与 `re.sub()` 函数相同，只不过 `re.subn()` 函数将返回一个元组用来保存替换的结果和替换次数。其函数原型如下：

```
re.sub( pattern, repl, string[, count])
re.subn( pattern, repl, string[, count])
```

其参数含义：

- `pattern` 正则表达式模式；
- `repl` 要替换成的内容；
- `string` 进行内容替换的字符串；
- `count` 可选参数，最大替换次数。

在交互式环境下使用上述函数进行内容替换的示例代码如下：

```
>>> import re                                # 导入 re 模块
>>> s = 'Life can be bad'                    # 定义字符串
```



```
>>> re.sub('bad','good',s)           # 用“good”替换“bad”
'Life can be good'
>>> re.sub('bad|be','good',s)        # 用“good”替换“bad”或者“be”
'Life can good good'
>>> re.sub('bad|be','good',s,1)      # 用“good”替换“bad”或者“be”，只替换一次
'Life can good bad'
>>> re.subn('bad|be','good',s,1)     # 用“good”替换“bad”或者“be”，只替换一次
('Life can good bad', 1)             # 返回由替换后的字符串和替换的次数组成的元组
>>> r = re.subn('bad|be','good',s)   # 用“good”替换“bad”或者“be”
>>> print(r[0])                      # 输出元组第一项
Life can good good
>>> print(r[1])                      # 输出元组第二项
2
```

13.2.3 split()函数

re.split()函数用于分割字符串，它返回分割后的字符串列表。其函数原型分别如下：

```
re.split( pattern, string[, maxsplit = 0])
```

其参数含义如下：

- pattern 正则表达式模式；
- string 要分割的字符串；
- maxsplit 可选参数，最大分割次数。



注意 该函数返回的数据类型为列表。

在交互式环境下使用上述函数对字符串进行分割操作的示例代码如下：

```
>>> import re                         # 导入re模块
>>> s = 'Life can be bad'             # 定义字符串
>>> re.split(' ',s)                   # 使用空格分割字符串（注：单引号之间有一个空格）
['Life', 'can', 'be', 'bad']
>>> r = re.split(' ',s,1)             # 只分割一次
>>> for i in r:                       # 遍历分割后返回的列表
... print(i)
...
Life
can be bad
>>> re.split('b',s)                  # 使用字母“b”分割字符串
['Life can ', 'e ', 'ad']
```

13.2.4 正则表达式对象

re 模块中包含一个 re.compile()函数，可以使用 re.compile()函数将正则表达式编译生成一个 RegexObject 对象实例。然后可以通过生成的 RegexObject 对象实例对字符串进行操作，如查找、替换等。对于多次使用的正则表达式，使用这种编译后的对象实例可以提高处理或匹配的速度。re.compile()的函数原型如下：

```
compile( pattern[, flags])
```

其参数含义如下：

- pattern 正则表达式的匹配模式；
- flags 可选参数，编译标志。

在交互式环境下编译生成一个 RegexObject 对象的示例代码如下：

```
>>> import re                         # 导入re模块
```

```
>>> re.compile('a*b',re.I|re.X)           # 编译正则表达式, 忽略大小写和模式中空格
<_sre.SRE_Pattern object at 0x011232F0>
# 使用 re.X 编译标志表示在匹配模式中忽略注释以及空格等字符
>>> re.compile(''
... \b                                # 匹配单词开始
... AA?                             # 以 A 或 AA 开头
... \d                               # 匹配一个数字 i
... \w*                             # 匹配任意字符
...                                # 一个空行
... \b                               # 匹配单词结束
... ''',re.X)
<_sre.SRE_Pattern object at 0x01093BD8>
```

正则表达式对象也是一个类, 具有自己的方法。正则表达式对象的方法与 `re` 模块中提供的函数基本相同。主要有以下几种。

1. match()

```
match( string[, pos[, endpos]])
```

其参数含义如下:

- `string` 要进行匹配的字符串;
- `pos` 可选参数, 进行匹配的起始位置;
- `endpos` 可选参数, 进行匹配的结束位置。

`match()`方法用于从字符串开始处进行匹配, 或者从指定位置处进行匹配。要匹配的字符串必须位于开始, 或者参数指定的位置才会匹配成功。如果匹配成功, `match()`返回一个 `MatchObject` 对象实例。

2. search()

```
search( string[, pos[, endpos]])
```

其参数含义如下:

- `string` 要进行匹配的字符串;
- `pos` 可选参数, 进行查找的起始位置;
- `endpos` 可选参数, 进行查找的结束位置。

对字符串进行查找, 不同的是 `search()`方法在整个字符串中搜索。如果查找成功, `search()`将返回一个 `MatchObject` 对象实例。正则表达式对象的 `findall()`方法用于在字符串中查找所有符合正则表达式的字符串, 并返回这些字符串的列表。

3. findall()

```
findall( string[, pos[, endpos]])
```

其参数含义与 `search()`方法相同。如果在正则表达式中使用了组, 则返回一个元组。

4. sub()和 subn()

```
sub( repl, string[, count = 0])
subn( repl, string[, count = 0])
```

其参数含义相同:

- `repl` 要替换成的内容;
- `string` 进行内容替换的字符串;
- `count` 可选参数, 最大替换次数。

它们主要用于对字符串的替换。



5. split()

```
split( string[, maxsplit = 0])
```

其参数含义如下：

- **string** 要分割的字符串；
- **maxsplit** 可选参数，最大分割次数。

正则表达式对象的 `split()` 方法用于对字符串进行分割。

以下代码在交互式环境下演示了应用正则表达式对象及其方法进行匹配、搜索与替换：

```
# 导入 re 模块
>>> import re
# 编译正则表达式，“go*d”表示在“g”和“d”之间有任何一个字母“o”的单词，
# 如“gd”，“god”，“good”
>>> r = re.compile('go*d')
# 在字符串开始处匹配，没有返回值，表示匹配失败
>>> r.match('Life can be good')
# 从字符串的第 13 个字符开始匹配（字符串从 0 开始），
# 也就是从字母“g”开始，返回 MatchObject 对象实例
>>> r.match('Life can be good',12)
<_sre.SRE_Match object at 0x01122640>
# 在字符串中搜索“go*d”，返回 MatchObject 对象实例，表示字符串中含有“go*d”
>>> r.search('Life can be good')
<_sre.SRE_Match object at 0x011226E8>
# 重新编译，匹配字母“b”和字母“g”之间包含一个字母以及一个空字符的情况
>>> r = re.compile('b.\sg')
# 在字符串中搜索，此处匹配的是“be g”
>>> r.search('Life can be good')
<_sre.SRE_Match object at 0x01122640>
# 重新编译，匹配任意两个字母后跟一个空字符和字母“g”的情况
>>> r = re.compile('\w.\sg')
# 在字符串中搜索，此处匹配的是“be g”
>>> r.search('Life can be good')
<_sre.SRE_Match object at 0x011227C8>
# 匹配后边有一个空字符的任意包含两个或者三个字符的单词
>>> r = re.compile('\b\w..\?\'s')
# 使用 findall() 方法查找
>>> r.findall('Life can be good')
['can ', 'be ']
>>> s = '''Life can be good;          # 定义字符串
... Life can be bad;
... Life is mostly cheerful;
... But sometimes sad.'''
>>> r = re.compile('b\w*',re.I)      # 编译正则表达式，忽略大小写
>>> new = r.sub('*',s)                # 使用 sub() 替换字符
>>> print(new)                        # 输出结果，可以看到所有以“b”开头的单词都被替换
Life can * good;
Life can * *;
Life is mostly cheerful;
* sometimes sad.
>>> new = r.sub('*',s,2)              # 只在字符串中替换两次
>>> print(new)
Life can * good;
Life can * bad;
Life is mostly cheerful;
But sometimes sad.
>>> r = re.compile('b\w*')           # 重新编译，不忽略大小写
>>> new = r.subn('*',s)               # 使用 subn() 替换字符，它返回一个元组
```

```

>>> print(new[0])
Life can * good;
Life can * *;
Life is mostly cheerful;
But sometimes sad.
>>> print(new[1])
3
>>> new = r.subn('*',s,1)
>>> print(new[0])
Life can * good;
Life can be bad;
Life is mostly cheerful;
But sometimes sad.
>>> print(new[1])
1
>>> s = '''Life can be good;
... Life can be bad;
... Life is mostly cheerful;
... But sometimes sad.'''
>>> r = re.compile('\s')
>>> news = r.split(s)
>>> print(news)
['Life', 'can', 'be', 'good;', 'Life', 'can', 'be', 'bad;', 'Life', 'is', 'mostly',
'cheerful;', 'But', 'sometimes', 'sad.']
>>> news = r.split(s,4)
>>> for new in news:
...     print(new)
...
Life
can
can
be
good;
Life can be bad;
Life is mostly cheerful;
But sometimes sad.
>>> r = re.compile('b\\w*',re.I)
>>> news = r.split(s)
>>> print(news)
['Life can ', ' good;\nLife can ', ' ', ';\nLife is mostly cheerful;\n', ' sometimes
sad. ']
>>> news = r.split(s,1)
>>> for new in news:
...     print(new)
...
Life can
good;
Life can be bad;
Life is mostly cheerful;
But sometimes sad.
>>> r = re.compile('\w*e',re.I)
>>> news = r.split(s)
>>> print(news)
['', ' can ', ' good;\n', ' can ', ' bad;\n', ' is mostly ', ' rful;\nBut ', 's sad. ']

```

输出替换后的字符串，可以看到“But”没有被替换

输出替换的次数

只在字符串中替换一次

定义字符串

编译匹配空字符的正则表达式

以空字符分割字符串

返回一个列表

只分割四次

遍历列表，输出分割后的字符串

编译匹配以字母“b”开头的字符串，忽略大小写

分割字符串返回列表

输出列表

只分割一次

遍历列表，输出字符串

编译匹配以字母“e”结尾的字符串，忽略大小写



注意

调用正则表达式对象的方法进行匹配时，需要先用 `compile()` 函数进行编译得到正则表达式对象。



13.3 分组匹配与匹配对象使用



在正则表达式中使用组，可以将正则表达式分解成几个不同的组成部分。在完成匹配或者搜索后，可以使用分组编号访问不同部分匹配的内容。

13.3.1 分组基础

在正则表达式中以一对圆括号 “()” 来表示位于其中的内容属于一个分组。例如 “(re)+” 将匹配 “rere” “rerere” 等多个 “re” 重复的情况。分组在匹配由不同部分组成的一个整体时非常有用。如电话号码由区号和号码组成，在正则表达式中可以使用两个分组来进行匹配：一个分组匹配区号；另一个分组匹配后边的号码。在交互式环境下演示代码如下：

```
>>> import re
>>> s = 'Phone No. 010-87654321'
>>> r = re.compile(r'(\d+)-(\d+)')
>>> m = r.search(s)
>>> m
<_sre.SRE_Match object at 0x01107E30>
>>> m.group(1)
'010'
>>> m.group(2)
'87654321'
>>> m.groups()
('010', '87654321')
```

在正则表达式中，可以通过使用 “(?P<组名>)” 为组设置一个名字，通过使用以下模式，将第一个组的名字设置为 “Area”，将第二组的名字设置为 “No”，如下所示：

```
r'(?P<Area>\d+)-(?P<No>\d+)'
```

上述的例子可以修改为：

```
>>> import re
>>> s = 'Phone No. 010-87654321'
>>> r = re.compile(r'(?P<Area>\d+)-(?P<No>\d+)')
>>> m = r.search(s)
>>> m
<_sre.SRE_Match object at 0x01122BA8>
>>> m.groupdict()
{'Area': '010', 'No': '87654321'}
>>> m.group('No')
'87654321'
>>> m.group('Area')
'010'
```

13.3.2 分组扩展

除了在组中使用 “(?P<组名>)” 来命名组名以外，还可以使用几种以 “?” 开头的扩展语法，如表 13.3 所示。

表 13.3 扩展语法及其意义

扩展语法	意 义
(?iLmsux)	设置匹配标志，可以是 i、L、m、s、u、x 以及它们的组合。其含义与编译标志相同
(?...)	匹配但不捕获该匹配的子表达式，即它是一个非捕获匹配，不存储，供以后使用的匹配
(?P=name)	表示在此之前的名为 name 的组
(?#...)	表示注释
(?=...)	用于正则表达式之后，表示如果 “=” 后的内容在字符串中出现则匹配，但不返回 “=” 后的内容



续表

扩展语法	意 义
(?!...)	用于正则表达式之后,表示如果“!”后的内容在字符串中不出现则匹配,但不返回“!”后的内容
(?<=...)	用于正则表达式之前,与(?=...)含义相同
(?<!...)	用于正则表达式之前,与(?!...)含义相同

上述模式在交互式环境下演示代码如下:

```
>>> import re                                # 导入 re 模块
>>> s = '''Life can be good;                 # 定义字符串
... Life can be bad;
... Life is mostly cheerful;
... But sometimes sad.
... '''
>>> r = re.compile(r'be(?:sgood)')          # 编译正则表达式,只匹配其后单词为“good”的“be”
>>> m = r.search(s)                          # 搜索字符串
>>> m                                         # 查看 m
<_sre.SRE_Match object at 0x0111ED40>
                                         # 返回一个 Matchobject 对象实例表示查找到单词
>>> m.span()                                # 输出匹配到的单词在字符串中的位置
(9, 11)
>>> r.findall(s)                             # 使用 findall() 方法输出所有匹配的单词
['be']
>>> r = re.compile('be')                    # 重新编译正则表达式,匹配单词“be”
>>> r.findall(s)                             # 使用 findall() 方法输出所有匹配的单词
['be', 'be']
>>> r = re.compile(r'be(?:!sgood)')         # 匹配之后单词不为“good”的“be”
>>> m = r.search(s)                          # 搜索字符串
>>> m                                         # 查看 m
<_sre.SRE_Match object at 0x010DAAD8>      # 返回一个 Matchobject 对象实例表示查找到单词
>>> m.span()                                # 输出匹配到的单词在字符串中的位置
(27, 29)
>>> r = re.compile(r'(?can\s)be(sgood)')    # 使用组来匹配“be good”
>>> m = r.search(s)
>>> m                                         # 查看 m
<_sre.SRE_Match object at 0x01112660>
>>> m.groups()                              # 使用 groups() 方法输出组
(' good',)
>>> m.group(1)                              # 使用组编号输出组
' good'
>>> r = re.compile(r'(?P<first>\w)(?P=first)') # 使用组名重复,此处匹配具有两个重复字母的单词
                                         # 使用组名重复,此处匹配具有两个重复字母的单词
>>> r.findall(s)                             # 输出匹配到的字母
['o', 'e']
>>> r = re.compile(r'(?can\s)b\w*\b')        # 匹配以字母“b”开头在“can”之后的单词
>>> r.findall(s)                             # 输出匹配到的单词
['be', 'be']
>>> r = re.compile(r'(?!can\s)b\w*\b')       # 匹配以字母“b”开头不在“can”之后单词
>>> r.findall(s)                             # 输出匹配到的单词
['bad']
>>> r = re.compile(r'(?!can\s)(i)b\w*\b')    # 重新编译,忽略大小写
>>> r.findall(s)                             # 输出匹配到的单词
['bad', 'But']
```

13.3.3 匹配对象与组的使用

Match 对象实例是由正则表达式对象的 match、search 方法在匹配成功后返回的。Match



对象有以下这几种常用的方法和属性，用于对匹配成功的正则表达式进行处理。

`group()`、`groups()`、`groupdict()`方法都是处理在正则表达式中使用“()”分组的情况。不同的是，`group()`的返回值为字符串，当传递多个参数时其返回值为元组；`groups()`的返回值为元组；`groupdict()`的返回值为字典。其原型分别如下：

```
group([group1, ...])
groups([default])
groupdict([default])
```

对于 `group()`，其参数为分组的编号。如果向 `group()`传递多个参数，则其返回各个参数所对应的字符串组成元组。对于 `groups()`和 `groupdict()`一般不需要向其传递参数。

以下在交互式环境下演示的实例使用了上述三种方法对字符串进行操作：

```
>>> import re                                # 导入 re 模块
>>> s = '''Life can be dreams,                # 定义字符串
... Life can be great thoughts;
... Life can mean a person,
... Sitting in a court.'''
>>> r = re.compile('\\b(?:P<first>\\w+)a(\\w+)\\b')

# 编译正则表达式，匹配所有包含字母“a”的单词

>>> m = r.search(s)                           # 从头开始搜索，返回搜索到的第一个单词
>>> m.groupdict()                             # 使用 groupdict() 输出字典
{'first': 'c'}

>>> m.groups()                                # 使用 groups() 输出元组
('c', 'n')

>>> m = r.search(s, 9)                         # 从指定位置开始重新搜索
>>> m.group()                                 # 输出匹配到的字符串
'dreams'

>>> m.group(1)                                # 输出第一对圆括号中的内容，即字母“a”之前部分
'dre'

>>> m.group(2)                                # 输出第二对圆括号中的内容，即字母“a”之后部分
'ms'

>>> m.group(1, 2)                             # 全部输出，返回一个元组
('dre', 'ms')

>>> m.groupdict()                             # 使用 groupdict() 输出字典
{'first': 'dre'}

>>> m.groups()                                # 使用 groups() 输出元组
('dre', 'ms')
```

13.3.4 匹配对象与索引使用

`start()`、`end()`、`span()`方法返回所匹配的子字符串的索引。其原型分别如下：

```
start([groupid=0])
end([groupid=0])
span([groupid=0])
```

其参数含义相同，`groupid` 为可选参数，即分组编号。如果不向其传递参数，则返回整个子字符串的索引。`start()`方法返回子字符串或者组的起始位置索引。`end()`方法返回子字符串或者组的结束位置索引。而 `span()`方法则以元组的形式返回以上两者。

其使用方法在交互式环境下演示代码如下：

```
>>> import re                                # 导入 re 模块
>>> s = '''Life can be dreams,                # 定义字符串
... Life can be great thoughts;
... Life can mean a person,
... Sitting in a court.'''
r = re.compile('\\b(?:P<first>\\w+)a(\\w+)\\b') # 编译正则表达式匹配含有字母“a”的单词
```

```

m = r.search(s,9)                # 从字符串中第 10 个字符开始搜索
>>> m.start()                  # 输出匹配到的子字符串的起始位置
12
>>> m.start(1)                  # 输出第一组的起始位置
12
>>> m.start(2)                  # 输出第二组的起始位置
16
>>> m.end(1)                    # 输出第一组的子字符串的结束位置
15
>>> m.end()                    # 输出子字符串的结束位置
18
>>> m.span()                   # 输出子字符串的开始和结束位置
(12, 18)
>>> m.span(2)                  # 输出第二组子字符串的开始和结束位置
(16, 18)

```

13.4 正则表达式应用示例

正则表达式是处理文本文件的强有力工具。本节给出了一个简单地使用正则表达式处理 Python 程序中的函数和变量的例子。

在 Python 程序中，函数定义必须以“def”开头，因此处理函数的过程相当简单。为了代码简洁，此处假设程序编写规范上在关键字“def”后跟一个空格，然后就是函数名，接着就是参数。没有考虑使用多个空格的情况。

而 Python 程序中的变量则不好处理，因为变量一般不需要事先声明，往往都是直接赋值。因此在程序中首先处理了变量直接赋值的情况。通过匹配单词后接“=”的情况查找变量名。同样，为了代码简洁，仅考虑比较规范整洁的写法，变量名与“=”之间有一空格。另外，还有一类变量是在 for 循环语句中直接使用的，因此程序中又特别处理了 for 循环的情况。为了使代码简洁，程序并没有处理变量名重复的情况。

【实例 13-1】演示了运用正则表达式分析并获取 Python 程序中定义的所有方法和变量，代码如下：

```

# -*- coding:utf-8 -*-
#
import re
import sys
def DealWithFunc(s):
    r = re.compile(r'''
        (?<=def\s)                # 前边必须含有 def 且 def 后跟一个空格
        \w+                        # 匹配函数名
        \(. *?\)                  # 匹配参数
        (?=:)                      # 后边必须跟一个 “:”
        ''',re.X | re.U)          # 设置编译选项，忽略模式中的注释
    return r.findall(s)
def DealWithVar(s):
    vars = []
    r = re.compile(r'''
        \b                          # 匹配单词开始
        \w+                          # 匹配变量名
        (?=\s=)                     # 处理为为变量赋值的情况
        ''',re.X | re.U)
    vars.extend(r.findall(s))
    r = re.compile(r'''
        (?<=for\s)                # 处理变量位于 for 语句中的情况
        \w+                          # 匹配变量名
        \s                          # 匹配空格
    ''',re.X | re.U)
    vars.extend(r.findall(s))

```



```

        (?=in)                                # 匹配 in
        '',re.X | re.U)                       # 设置编译选项, 忽略模式中的注释
    vars.extend(r.findall(s))
    return vars

# 判断命令行是否有输入, 没有则要求输入要处理的文件
if len(sys.argv) == 1:
    sour = input('请输入要处理的文件路径')
else:
    sour = sys.argv[1]
file = open(sour,encoding="utf-8") # 打开文件
s = file.readlines()              # 将文件内容以行读入的 s 中
file.close()                      # 关闭文件
print('*****')
print(sour,'中的函数有: ')
print('*****')
i = 0                              # i 为函数所在的行号
# 循环处理每一行, 匹配其中的函数并输出函数所在的行号以及函数的原型
for line in s:
    i = i + 1
    function = DealWithFunc(line)
    if len(function) == 1:
        print('Line: ',i,'\t',function[0])
print('*****')
print(sour,'中的变量有: ')
print('*****')
i = 0                              # 此处 i 为变量所在的行号
# 循环处理每一行, 匹配其中的变量, 输出变量所在的行号以及变量名
for line in s:
    i = i + 1
    var = DealWithVar(line)
    if len(var) == 1:
        print('Line: ',i,'\t',var[0])

```

【代码说明】代码中定义了一个用于获取函数名的函数 DealWithFunc()和一个用于获取变量名的函数 DealWithVar()。之后分别调用它们进行查找和获取。

【运行效果】程序运行后, 输入本程序文件名, 将输出如图 13.1 所示的结果。

```

>>>
请输入要处理的文件路径: a13_1.py
*****
a13_1.py 中的函数有:
*****
Line: 5      DealWithFunc(s)
Line: 13     DealWithVar(s)
*****
a13_1.py 中的变量有:
*****
Line: 6      r
Line: 14     vars
Line: 15     r
Line: 21     r
Line: 30     sour
Line: 32     sour
Line: 33     file
Line: 34     s
Line: 39     i
Line: 40     line
Line: 41     i
Line: 42     function
Line: 48     i
Line: 49     line
Line: 50     i
Line: 51     var

```

图 13.1 正则表达式获取函数名与变量名

13.5 小结

正则表达式的功能非常强大,学习难度也较大,本章以尽可能多的操作代码演示了正则表达式的用法。首先介绍了正则表达式的基本元字符、常用正则表达式分析。其次介绍了怎样使用 Python 的 re 模块处理正则表达式,如用 match 函数进行搜索、使用 sub 函数进行内容替换、使用 split 函数分割等。接着介绍了将正则表达式编译为对象,以提供更高性能的方法。还介绍了正则表达式中的分组、匹配和搜索的结果对象——Match 对象的使用等内容。以后要多编写代码进行学习、验证。在其他程序设计语言中也可以直接使用在这里学习的正则表达式。

13.6 本章习题

一、简答题

1. 正则表达式的主要作用是什么?
2. 在 Python 中使用正则表达式有哪两种使用方法?
3. 请你写出以下要求的正则表达式?
 - (1) 匹配和区分电话号码(格式为 010-23452341)手机号码
 - (2) 测试用户输入是否为电子邮箱的格式字符串
 - (3) 匹配身份证号(包含 15 位和 18 位)
 - (4) URL 地址(http://www.abc.com)
 - (5) IP 地址
4. tkinter 中布局组件有哪几种?其布局的方法是怎样的?

二、实验题

1. 编程实现用正则表达式的方法来匹配获取以下 HTML 代码中的所有链接地址:

```
<h3>联系我们</h3>
<p>联系人: 王经理</p>
<p>电话: 021-87017800</p>
  <div id="nav">
    <ul>
      <li><a class="nav-first" href="/">首 页</a></li>
      <li><a href="/lista.php">吸粮机</a></li>
      <li><a href="/listb.php">灌包机</a></li>
      <li><a href="/listc.php">汽油吸粮机</a></li>
      <li><a href="/order/setorder.php">我要订购</a></li>
      <li><a href="/about.php">关于我们</a></li>
    </ul>
  </div>
```

2. 现有某日志系统生成的日志文本文件,其内容模式为:

```
[info] 2015-07-30 15:54:03 asd.py[line:12] program start.
[debug] 2015-07-09 11:54:12 asd.py[line:12] generate uid 2354.
[warning] 2015-07-12 23:51:03 asd.py[line:12] uid is used up.
```

请编程实现过滤掉所有 info、debug 的信息,将 warning 信息的时间、文件名及相关信息保存在文件 warning.txt 中。

第 14 章 网络编程

网络编程是现代编程主题中的一个重要组成部分，而 Python 在标准库中就已经提供了丰富的网络编程模块，以支持用户进行编写具有各种网络功能的程序或软件。在 Python 标准库中，支持底层网络编程的是 socket 模块；针对特定的网络协议进行编程的模块有 urllib、http、ftplib、poplib、smtpplib、telnetlib、socketserver 等，还有 ipaddress 等工具模块。此外，进行网络编程的还有著名的第三方模块 Twisted，用来进行异步网络编程，也是极好的工具，不过此时只支持 Python2，相信未来也会支持 Python3 版本的。

Python 网络编程既可以实现服务器端编程，也可以实现客户端编程。本章内容包括：

- 网络编程预备知识；
- 用 socket 建立客户端与服务器；
- 用 socketserver 建立基本的服务器；
- 使用 http、urllib 标准库；
- 用 poplib 与 smtpplib 处理邮件；
- 用 ftplib 访问 FTP 服务器。

14.1 网络编程基础



作为网络程序的开发者，无论开发的是服务器端程序还是客户端程序，都需要扎实的计算机网络方面的基础知识，这样才能在网络编程时游刃有余，此处仅对计算机网络相关知识作出必要的介绍，详细的内容应参考相关资料。

14.1.1 什么是网络

计算机网络是一些相互连接的自主计算机或设备的集合，它是计算机技术和通信技术相结合的产物。这些计算机之间的连接可以使用任何一种能够通信的介质，比如网线、光纤、微波、红外线，甚至通信卫星。而自主计算机或设备可以是如大型计算机、微型计算机、笔记本电脑、平板电脑、手机、路由器、调制解调器（俗称猫）等。

计算机网络根据其覆盖的地域范围的大小可分为局域网、城域网和广域网。而其中局域网（LAN）的作用范围最小，一般为一个单位、一幢楼、一间办公室或一个家庭的网络，有的甚至只有两台计算机；城域网（MAN）的作用范围一般是一个城市或几个街区等，可以为一个或几个单位拥有，也可以是公用的，能将多个局域网互联。广域网（WAN）的作用范围更大，可以是一个省、一个国家甚至是全球。现在看到的网络大多是相互连接在一起的，就形成了互联网络，即互联网（Internet）。

14.1.2 网络协议

网络协议是网络中进行数据交换与传输所需要的规则、标准或约定，主要由语法（数据与信息结构形式）、语义和同步（事件的实现顺序）三个要素组成。

世界上最先提出的协议理论模型是由国际标准化组织（ISO）提出的开放系统互联基本参考模型（OSI），它采用的是七层协议的体系结构。虽然 OSI 清晰完整，但终因复杂又不实用而

没有得到使用。另一方面,使用了简化的 OSI 的 TCP/IP 协议却得到了非常广泛的应用,它是一个四层的体系结构,包括应用层、运(传)输层、网际(络)层和网络接口层。

如图 14.1 所示, TCP/IP 协议其实是一个协议簇,不仅包括 TCP 和 IP 协议,还包括 UDP、FTP、HTTP、SMTP 等,此外还包括一些图中没有显示的 ICMP、ARP、RARP 等协议。



图 14.1 TCP/IP 协议图

这种分层的协议结构还表示出,上层协议需要传输的数据,应该交给它紧邻的下层。而应用层和传输层分别有两个以上协议,所以对于应用层来说,不同的协议数据可以通过传输层的不同协议来传输。例如同是文件传输协议,FTP 协议在传送数据时就使用下层的 TCP 协议,而 TFTP 协议使用下层的 UDP 协议进行数据传输。



在实际的网络程序开发中,必须要对其使用的协议有相关的理解。

14.1.3 地址与端口

在互联网上同时连接的计算机数量庞大,要想与某一台计算机进行数据传输,就必须要先找到对方计算机。在 TCP/IP 协议中的网络层的 IP 协议提供了网络上的计算机地址系统,即 IP 地址。

目前广泛使用的 IP 地址是 IPv4 版本的地址,使用 32 位二进制代码表示的网络地址,为了方便人们使用,将 32 位二进制代码划分为 4 个 8 位的二进制代码,并将其转换为十进制数,中间用点来分开,称为点分十进制表示法。如图 14.2 所示,实际的 IP 地址 01111101000011011001101000110101 用点分十进制表示法则为 125.13.154.53。

01111101	00001101	10011010	00110101
125	13	154	53

图 14.2 IP 地址转换

在同一台计算机中可以同时运行多个网络程序,协议又如何区分这不同的网络程序所传输的数据呢?这涉及端口的使用。TCP/IP 协议规定端口值范围为 0~65535,共 65536 个。而在同一台计算机上,TCP 和 UDP 都有自己的端口范围,其范围也相同且并不冲突。

端口在 TCP/IP 协议中共分为三类:

- 0~1023 称为周知端口,一般都有固定的协议使用;
- 1024~49151 称为注册端口,程序员可以自由注册使用;
- 49152~65535 称为动态端口,由操作系统动态分配。



比如常用的 FTP 协议端口号为 21, HTTP 常用端口为 80 等。而程序员开发期间可以根据自己开发的应用和所用计算机的情况来使用端口。如开发 WEB 服务器,也可以使用 80 端口(计算机没有使用 80 端口服务),但一般在开发阶段会使用上述分类中的注册端口,待完成后部署时,改为 80 端口也可以。

因此,在编写服务器端程序时,应该指定服务的 IP 地址与端口;而在编写客户端程序时,应该指定要连接的服务器的 IP 地址与端口。而在开发阶段,由于调试程序可能只在开发者的一台计算机上同时运行服务器与客户端,这时可以指定一个特殊的 IP 地址,即 127.0.0.1(回环地址)或直接用字符串 localhost 来代表本机。当程序实际部署时,应根据所部署的计算机来指定 IP 地址与端口即可。



要注意 要学好网络编程,必须搞清楚 IP 地址与端口概念及应用。

读者若要了解 TCP/IP 协议更详细的知识,可以参阅相关资料或书籍,这里不做详解。

14.2 套接字的使用



TCP/IP 协议中的 TCP 和 UDP 协议都通过一种套接字(socket)来实现网络功能。套接字是一种类文件对象,它使程序能接受客户端的连接或建立对客户端的连接,用以发送和接收数据。不论是客户端程序还是服务器端程序,为了进行网络通信,都要创建套接字对象。

14.2.1 用 socket 建立服务器端程序

在 Python 标准库中,使用 socket 模块中提供的 socket 对象,就可以在计算机网络中建立服务器与客户端,并且能够进行通信。服务器端需要建立一个 socket 对象,并等待客户端的连接。客户端使用 socket 对象与服务器端进行连接,一旦连接成功,客户端和服务器端就可以进行通信了。

socket 模块中的 socket 对象是 socket 网络编程的基础对象,其初始化原型如下:

```
socket( family, type, proto)
```

其参数含义如下所示。

- family: 地址族,可选参数。默认为 AF_INET(IPv4),也可以是 AF_INET6 或 AF_UNIX;
- type: socket 类型,可选参数。默认为 SOCK_STREAM(TCP 协议),可用 SOCK_DGRAM(UDP 协议);
- proto: 协议类型,可选参数。默认为 0。



要注意 以上只列出常用参数,其他参数可以参考相关资料。

作为服务器端的 socket 对象主要应用以下这些常用的方法。

1. bind(address)

其参数 address 是由 IP 地址和端口组成的元组,例如“(‘127.0.0.1’,1051)”。如果 IP 地址为空,则表示本机。它的作用是使 socket 和服务器服务地址相关联。

2. listen(backlog)

参数 backlog 指定在拒绝连接之前,操作系统允许它的最大挂起连接数量。最小值为 0(如果用户使用了更小的值,则会自动被置为 0),大部分程序最多设置为 5 就足够了。

该方法将 socket 设置为服务器模式，之后就可以调用以下的 accept() 方法等待客户端的连接。

3. accept()

它会等待进入的连接，并返回一个由新建的与客户端的 socket 连接和客户端地址组成的元组，而客户的地址也是一个由客户端 IP 地址和端口组成的元组。

4. close()

显而易见，这个方法的作用就是关闭该 socket，停止本程序与服务器或客户端的连接。

5. recv(buffer size[, flag])

用于接收远程连接发来的信息，并返回该信息。buffer size 可以设定缓冲区的大小。

6. send(data[, flags])

用于向连接的远端发送信息，data 应该是 bytes 类型的数据。其返回值为已传送的字节数。而建立服务器端的 socket 就要依次使用这几个方法，其基本顺序如图 14.3 所示。

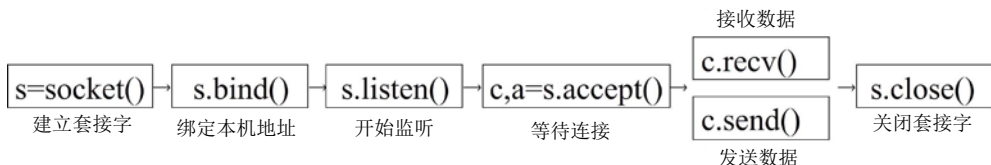


图 14.3 socket 服务器建立流程图

【实例 14-1】演示了以 TCP 连接方式使用 socket 建立一个简单的服务器端程序，基本功能是将收到的信息直接发回客户端，代码如下：

```
import socket

HOST = ''
PORT = 10888
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
conn, addr = s.accept()
print('Client\'s Address:', addr)
while True:
    data = conn.recv(1024)
    if not data:
        break
    print("Receive Data:", data.decode('utf-8'))
    conn.send(data)
conn.close()
```

【代码说明】对比如图 14.3 所示的 socket 服务器建立流程图可以看出，其基本流程是相同的。在建立连接之后，程序使用了一个 while 语句来多次与客户端交换数据，直到收到数据为空，就会中止服务器的运行。

【运行效果】由于这是一个服务器端程序，运行之后程序不会立即返回，而是等待客户端连接，所以现在还看不出什么效果。等之后建立客户端才能看到具体效果。

14.2.2 用 socket 建立客户端程序

相比用 socket 建立服务器端而言，建立客户端程序要简单得多。当然还是需要创建一个



socket 的实例，而后调用这个 socket 实例的 connect() 方法来连接服务器端即可。这个方法原型如下：

```
connect(address)
```

参数 address 通常也是一个元组（由一个主机名/IP 地址，端口构成），当然要连接本地计算机的话，主机名可直接使用 'localhost'，它用于将 socket 连接到远程以 address 为地址的计算机。

用 socket 建立客户端程序的基本流程如图 14.4 所示。

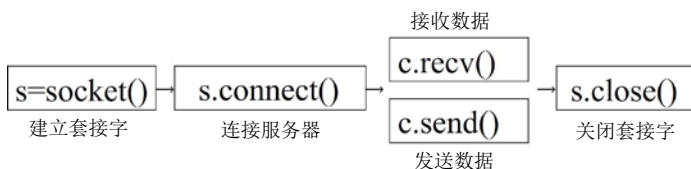


图 14.4 用 socket 建立客户端流程图

【实例 14-2】演示了配合实例 14-1 以 TCP 连接方式使用 socket 建立一个简单的客户端程序，基本功能是从键盘录入的信息发送给服务器，并从服务器接收信息，代码如下：

```
import socket

HOST = 'localhost'
PORT = 10888
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
data = "你好!"
while data:
    s.sendall(data.encode('utf-8'))
    data = s.recv(512)
    print("Receive from server:\n",data.decode('utf-8'))
    data = input('please input a info:\n')
s.close()
```

【代码说明】实例 14-1 的服务器端建立在 'localhost' 的 10888 端口上，所以本例作为其客户端程序，连接的就是 'localhost' 的 10888 端口。当连接成功，向服务器端发送了一个默认的信息“你好！”之后，便从键盘录入信息向服务器端发送，直到录入空信息（直接敲回车），退出 while 循环，关闭 socket。

【运行效果】TCP 服务器与客户端的运行结果如图 14.5 所示，在命令提示符中先运行 a14_1.py 服务器端程序，然后运行 a14_2.py 客户端程序，除了发送一个默认的顺序外，从键盘录入的信息都会发送给服务器，服务器收到后显示并再次转发回客户端进行显示。

<pre>D:\>python a14_1.py Client's Address: ('127.0.0.1', 1048) Receive Data: 你好! Receive Data: abc Receive Data: def Receive Data: D:\></pre>	<pre>D:\>python a14_2.py Receive from server: 你好! please input a info: abc Receive from server: abc please input a info: def Receive from server: def please input a info:</pre>
---	---

图 14.5 TCP 服务器与客户端的运行结果

14.2.3 用 socket 建立基于 UDP 协议的服务器与客户端程序

通过使用 socket 应用传输层的 UDP 协议建立服务器与客户程序，从步骤上来看，要比使用 TCP 协议还要简单一点。发送和接收数据使用 socket 对象的主要方法如下：

```
recvfrom(bufsize[, flags])    # bufsize 用来指定缓冲区大小
```

主要用来从 socket 接收数据，该方法用于连接 UDP 协议：

```
sendto(bytes, address)
```

参数 bytes 是要发送的数据，address 是发送信息的目标地址，仍然是由目录 IP 地址和端口构成的元组。主要用来通过 UDP 协议将数据发送到指定的服务器端。

用 socket 建立基于 UDP 协议的服务器流程如图 14.6 所示。

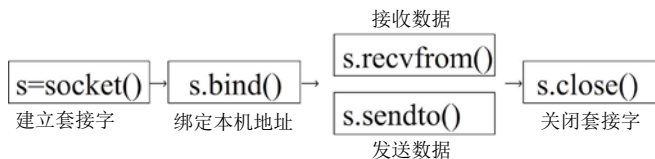


图 14.6 UDP 协议的服务器流程图

用 socket 建立基于 UDP 协议的服务器流程，如图 14.7 所示。

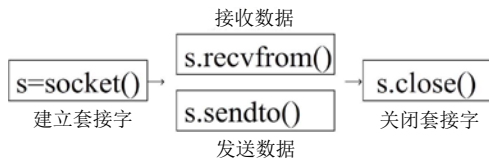


图 14.7 UDP 协议的客户端流程图

由以上两个流程图可以看出，基于 UDP 协议的服务器与客户端在进行数据传送时，不是先建立连接，而是直接进行数据传送。



注意 使用 UDP 协议进行网络编程和使用 TCP 协议进行网络编程的区别与联系，以及它们的应用场合也是不同的。

【实例 14-3】 演示了用 socket 建立基于 UDP 协议的服务器程序的实例代码，如下：

```
import socket

HOST = ''
PORT = 10888
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind((HOST, PORT))
data = True
while data:
    data, address = s.recvfrom(1024)
    if data == b'bye':
        break
    print('Received String:', data.decode('utf-8'))
    s.sendto(data, address)
s.close()
```

【代码说明】 实例中服务器程序建立在本机的 10888 端口，当收到 'bye' 时退出 while 循环，关闭服务器。



【实例 14-4】演示了用 socket 建立基于 UDP 协议的客户端程序的实例代码，如下：

```
import socket

HOST = 'localhost'
PORT = 10888
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
data = "你好!"
while data:
    s.sendto(data.encode('utf-8'),(HOST,PORT))
    if data=='bye':
        break
    data,addr = s.recvfrom(512)
    print("Receive from server:\n",data.decode('utf-8'))
    data = input('please input a info:\n')
s.close()
```

【代码说明】客户端创建 socket 后，直接向服务器端（本机的 10888 端口）发送数据，而没有进行连接。当用户键入'bye'时退出 while 循环，关闭本程序。

【运行效果】基于 UDP 协议的服务器与客户端的运行结果如图 14.8 所示，其基本效果与图 14.5，即 TCP 服务器与客户端的运行结果基本相同。

<pre>D:\lx>python a14_3.py Received String: 你好! Received String: abc Received String: def D:\lx></pre>	<pre>D:\lx>python a14_4.py Receive from server: 你好! please input a info: abc Receive from server: abc please input a info: def Receive from server: def please input a info: bye</pre>
---	---

图 14.8 基于 UDP 协议的服务器与客户端的运行结果

14.2.4 用 socketserver 模块建立服务器

虽然使用 socket 模块可以创建服务器，但是程序员要对包括网络连接等进行管理和编程。为了更加方便地创建网络服务器，Python 标准库中提供了一个创建网络服务器的框架——socketserver。

socketserver 框架将处理请求划分为两个部分，分别对应服务器类和请求处理类。服务器类处理通信问题，请求处理类处理数据交换或传送。这样，更加容易进行网络编程和程序的扩展。同时，该模块还支持快速的多线程或多进程的服务器编程。

socketserver 模块中使用的服务器类主要有 TCPServer、UDPServer、ThreadingTCPServer、ThreadingUDPServer、ForkingTCPServer、ForkingUDPServer 等。其中有 TCP 字符的就是使用 TCP 协议的服务器类，有 UDP 字符的就是使用 UDP 协议的服务器类，有 Threading 字符的是多线程服务器类，有 Forking 字符的是多进程服务器类。要创建不同类型的服务器程序，只需继承其中之一或直接实例化，然后调用服务器类方法 serve_forever()。这些服务器的构造方法参数主要有：

```
server_address      # 由 IP 地址和端口构成的元组
RequestHandlerClass # 处理器类，供服务器类调用处理数据
```

socketserver 模块中使用的处理器类主要有 StreamRequestHandler（基于 TCP 协议的）和

DatagramRequestHandler (基于 UDP 协议的)。只要继承其中之一, 就可以自定义一个处理器类。通过覆盖以下三个方法, 实现自定义:

- setup() 为请求准备请求处理器 (请求处理的初始化工作);
- handler() 完成具体的请求处理工作 (解析请求、处理数据、发出响应);
- finish() 清理请求处理器相关数据。

一般地说, 自定义一个简单的请求处理器, 只需覆盖 handler() 方法。

【实例 14-5】演示了用 socketserver 模块来实现实例 14-1 的基于 TCP 协议的服务器, 代码如下:

```
import socketserver

HOST = 'localhost'
PORT = 10888

class MyTcpHandler(socketserver.StreamRequestHandler):
    def handle(self):
        while True:
            data = self.request.recv(1024)
            if not data:
                Server.shutdown()
                break
            print('Receive Data:', data.decode('utf-8'))
            self.request.send(data)
        return

Server = socketserver.TCPServer((HOST, PORT), MyTcpHandler)
Server.serve_forever()
```

【代码说明】实例中自定义了一个继承自 StreamRequestHandler 的处理器类, 并覆盖了 handler() 方法, 以实现数据处理。然后直接实例化了 TCPServer 类, 调用 serve_forever() 方法启动服务器。该实例可以使用实例 14-2 的客户端来与这个服务器程序交互。其运行效果参见图 14.5 (TCP 服务器与客户端的运行结果)。



注意

实例中也可以使用 self.rfile.readline() 和 self.wfile.write(data) 这两个类文件对象进行收发数据。但客户端每次发送数据要有行结束符。

14.3 urllib 与 http 包使用

Python 标准库中的 socket 模块主要应用于底层网络协议, 编写程序要从底层开始构建, 要自行处理相关协议比较麻烦。其实对于大多数程序员来说, 网络编程都是针对应用协议进行的。比如本节所述的 urllib 与 http 包。

14.3.1 urllib 和 http 包简介

urllib、http 主要是面向 HTTP 协议的, 而网络上的网站应用都是基于 HTTP 协议, 因此使用它们编程可以轻松访问网站。其中 urllib 主要用于处理 URL (Universal Resource Locators), 使用 urllib 操作 URL 可以像使用和打开本地文件一样, 非常简单而又容易上手。http 则实现了对 HTTP 协议的封装, 是 urllib.request 的底层。

1. urllib 包简介

urllib 包的主要模块有:



- `urllib.request` 用于打开 URL 网址;
- `urllib.error` 定义了常见的 `urllib.request` 会引发的异常;
- `urllib.parse` 用于解析 URL;
- `urllib.robotparser` 用于解析 `robots.txt` 文件。

`urllib.request` 模块中提供了用于打开一个 URL 的函数 `urlopen()`, 其原型如下:

```
urlopen(url, data, proxies)
```

其参数含义如下:

- `url` 要进行操作的 URL 地址;
- `data` 向 URL 传递的数据, 可选参数;
- `proxies` 使用的代理地址, 可选参数。

`urlopen` 将返回一个 `HTTPResponse` 实例 (类文件对象), 可以像操作文件一样使用 `read`、`readline`、`close` 等方法对 URL 进行操作。

使用 `urllib.request` 模块中的 `urlretrieve` 方法可以将 URL 保存为本地文件。`urlretrieve` 方法的原型如下:

```
urlretrieve(url, filename, reporthook, data)
```

其参数含义如下:

- `url` 要保存的 URL 地址;
- `filename` 指定保存的文件名, 可选参数;
- `reporthook` 回调函数, 可选参数;
- `data` 发送的数据, 一般用于 `POST`, 可选参数。

在 `urllib.parse` 模块中还有一个可以对 URL 进行编码的函数 `urlencode()`, 其原型如下:

```
urlencode(query, doseq)
```

其参数含义如下:

- `query` 要进行编码的变量和值组成的字典;
- `doseq` 可选参数, 若为 `True` 则将元组的值分别编码成 “变量=值” 的形式。

使用 `urllib.parse` 模块中的 `quote` 方法和 `quote_plus` 方法可以替换字符串中的特殊字符, 使其符合 URL 所要求使用的字符。这两个方法的原型如下:

```
quote(string, safe)
quote_plus(string, safe)
```

这两个方法的参数相同, 含义如下:

- `string` 要进行替换的字符串;
- `safe` 可选参数, 指定不需要替换的字符, 默认为 “/”。

使用 `urllib.parse` 模块中的 `unquote` 方法和 `unquote_plus` 方法可以将使用 `quote` 方法和 `quote_plus` 方法替换后的字符还原 (在 Python 2.x 中直接由 `urllib` 模块进行处理)。这两个方法的原型如下:

```
unquote(string)
unquote_plus(string)
```

其参数 `string` 就是要进行还原的字符串。

此外, 这个包中还提供了一些处理 URL 所需要的如认证处理类、cookie 处理类等一些打开 URL 所需的基类。

2. http 包简介

http 包提供了使用 HTTP 协议的一些功能，其主要模块如下：

- `http.client` 底层的 HTTP 协议客户端，可以为 `urllib.request` 模块所用；
- `http.server` 提供了基于 `socketserver` 模块的基本 HTTP 服务器类；
- `http.cookies` `cookies` 的管理工具；
- `http.cookiejar` 提供了 `cookies` 的持久化支持。

在 `http.client` 模块中用于客户端的类如下所示：

- `HTTPConnection` 基于 HTTP 协议的访问客户端；
- `HTTPSConnection` 基于 HTTPS 协议的访问客户端；
- `HTTPResponse` 基于 HTTP 协议的服务端回应。

本小节主要介绍 `HTTPConnection` 与 `HTTPResponse`。

(1) `HTTPConnection` 构造方法的原型为：

```
HTTPConnection(host, port=None, [timeout, ]source_address=None)
```

其基本参数意义如下：

- `host` 服务器地址（可以使用 `www.abc.com:8080` 模式）；
- `port` 用来指定访问的服务器端口，不提供的话则从 `host` 提取，否则使用 80 端口；
- `timeout` 指定超时秒数。

`HTTPConnection` 对象的主要方法如下：

```
request(method, url, body, headers)
```

其参数含义如下：

- `method` 发送的操作，一般为“GET”或“POST”；
- `url` 进行操作的 URL；
- `body` 发送的数据；
- `headers` 发送的 HTTP 头。

当向服务器发送请求后，可以使用 `HTTPConnection` 对象的 `getresponse()` 方法返回一个 `HTTPResponse` 对象。使用 `HTTPConnection` 对象的 `close()` 方法可以关闭同服务器的连接。除了使用 `request` 方法以外，还可依次使用以下方法向服务器发送请求：

```
putrequest(request, selector, skip_host, skip_accept_encoding)
putheader(header, argument, ...)
endheaders()
send(data)
```

`putrequest` 方法的参数含义如下：

- `request` 所发送的操作；
- `selector` 进行操作的 URL；
- `skip_host` 可选参数，若为真，禁止自动发送“HOST:”；
- `skip_accept_encoding` 可选参数，若为真，禁止自动发送“Accept-Encoding:headers”。

`putheader` 方法的参数含义如下：

- `header` 发送的 HTTP 头；
- `argument` 发送的参数。

`send` 方法的参数含义如下：

- `data` 发送的数据。

(2) `HTTPResponse` 对象主要用于处理服务器对所发送请求的响应。使用 `HTTPResponse`



对象的 `read()` 方法可以获得服务器响应主体，也可以使用 `readline()` 方法每次读取一行，还可以用 `readlines()` 返回一个行列表。使用 `HTTPResponse` 对象的 `getheader()` 方法可以获得服务器响应的 HTTP 头，其原型如下：

```
getheader(name, default)
```

其参数含义如下：

- `name` 指定 HTTP 头名；
- `default` 可选择参数，如果指定的 `name` 不存在，则获取 `default` 指定的 HTTP 头。

`getheaders()` 返回 (header, value) 元组构成的列表

`HTTPResponse` 对象还具有 `version`、`status` 和 `reason` 等属性，用于查看 HTTP 协议的版本、状态等。

14.3.2 用 urllib 和 http 包访问网站

`urllib.request` 是 `http.client` 的抽象，要访问网站，可以使用 `urllib.request.urlopen()`，只需要一行代码，非常简单。当然也可以使用 `http.client.HTTPConnection` 对象。

【实例 14-6】 演示了使用 `urlopen()` 在百度搜索关键词得到第一页链接，代码如下：

```
from urllib.request import urlopen
from urllib.parse import urlencode
import re

##wd = input('输入一个要搜索的关键词: ')
wd= 'python'
wd = urlencode({'wd':wd})
url = 'http://www.baidu.com/s?' + wd
page = urlopen(url).read()
content = (page.decode('utf-8')).replace("\n","").replace("\t","")
title = re.findall(r'<h3 class="t".*?h3>', content)
title = [item[item.find('href =')+6:item.find('target=')]] for item in title]
title = [item.replace(' ','').replace("'",'') for item in title]
for item in title:
    print(item)
```

【代码说明】 实例中用 `urlencode()` 对搜索的关键词进行 URL 编码，然后拼接到百度的网址后，应用 `urlopen()` 发出请求并取得结果，最后通过将结果进行解码和正则搜索与字符串处理后输出。如果将程序中的注释去除而把其后一句注释掉，就可以在运行时自主输入搜索的关键词。

【运行效果】 本例程序的运行效果如图 14.9 所示（只截取部分）：

```
>>>
http://www.baidu.com/link?url=GdRF9IHt2zqUAMS0wGevisljWoJ86rMpkDVOOBsLUrK
http://www.baidu.com/link?url=bYglKO0QM00M3hJeo4UWutDvHus2ECw3tyDigz0siVvK_Za-5
dCZWcV5A_e49EA9Q27GYfHmcqYUXUsxnnX_s8nPcecopFzB92MaNQ8KRszat7mmSx5B0cCVoWWWh7sR
http://www.baidu.com/link?url=pyE8CIOKX6UCFlvYzMASxWXfJe-Q-4yoT689zrcBsjdPz6PMA1
b6WHePIYo35qA5
```

图 14.9 通过百度搜索的超链接结果

【实例 14-7】 演示了应用 `http.client.HTTPConnection` 对象访问网站，代码如下：

```
from http.client import HTTPConnection

mc = HTTPConnection('www.baidu.com:80')
mc.request('GET', '/')
res = mc.getresponse()
print(res.status,res.reason)
```

```
print(res.read().decode('utf-8'))
```

【代码说明】这只是一个基本的访问实例，首先实例化 `http.client.HTTPConnection` 对象，指定请求的方法为 `GET`，最后用 `getresponse()` 方法取得访问的网页，并打印出响应的状态与网页。

【运行效果】访问百度首页输出（部分）如图 14.10 所示。

```
200 OK
<!DOCTYPE html><!--STATUS OK-->
<html>
<head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=Edge">
    <link rel="dns-prefetch" href="//s1.bdstatic.com"/>
    <link rel="dns-prefetch" href="//t1.baidu.com"/>
    <link rel="dns-prefetch" href="//t2.baidu.com"/>
```

图 14.10 访问百度首页输出（部分）

对比实例 14-6 可以看出，简单的网站访问使用 `urlopen()` 方法最简便。

14.4 用 poplib 与 smtplib 库收发邮件

Python 中的 `poplib` 模块和 `smtplib` 模块提供了对 POP3 协议和 SMTP 协议的支持，使用 POP3 协议可以登录 Email 服务器收取邮件，使用 SMTP 协议则可通过 Email 服务器发送邮件。

14.4.1 用 poplib 检查邮件

一般的邮箱服务都提供了 POP3 收取邮件的方式，Outlook 等 Email 客户端就是使用 POP3 协议收取邮箱中的邮件。使用 Python 的 `poplib` 模块可以编写一个简单地检查邮件的客户端脚本。

1. poplib 模块简介

使用 `poplib` 模块中的 `POP3` 类可以创建一个 POP3 对象实例。其原型如下：

```
POP3(host, port)
```

其参数含义如下：

- `host` POP3 邮件服务器；
- `port` 服务器端口，可选参数，默认为 110。

当创建一个 POP3 对象实例后可以使用其 `user` 方法向 POP3 服务器发送用户名。其原型如下：

```
user(username)
```

其参数含义如下：

- `username` 登录服务器的用户名。

使用 POP3 对象的 `pass_` 方法（注意 `pass` 后面有一个下划线字符）可以向 POP3 服务器发送密码。其原型如下：

```
pass_(password)
```

其参数 `password` 是指登录服务器的密码。

当登录服务器后，可以使用 POP3 对象的 `getwelcome` 方法获取服务器的欢迎信息。使用 POP3 对象的 `set_debuglevel` 方法可以设置调试级别。其原型如下：

```
set_debuglevel(level)
```

其参数 `level` 是指调试级别，用以显示与邮件服务器交互的相关信息。



使用 POP3 对象的 `stat` 可以获取邮箱的状态，如邮件数、邮箱大小等。使用 POP3 对象的 `list` 方法可以获得邮件内容列表，其原型如下：

```
list(which)
```

其参数 `which` 为可选参数，如果指定则仅列出指定的邮件内容。

使用 POP3 对象的 `retr` 方法可以获取指定的邮件。其原型如下：

```
retr(which)
```

其参数 `which` 表示指定要获取的邮件。

使用 POP3 对象的 `dele` 方法可以删除指定的邮件。其原型如下：

```
dele(which)
```

其参数 `which` 是指定要删除的邮件。

使用 POP3 对象的 `top` 方法可以收取邮件部分内容。其原型如下：

```
top(which, howmuch)
```

- `which` 指定获取的邮件；
- `howmuch` 指定获取的行数。

此外，还可使用 POP3 对象的 `rset` 方法清除收件箱中邮件的删除标记；使用 POP3 对象的 `noop` 方法保持同服务器的连接；使用 POP3 对象的 `quit` 方法断开同服务器的连接。

2. 使用 Python 检查 Email

使用 Python 检查 Email，首先应该知道自己所使用的 Email 的 POP3 服务器地址和端口。一般来说，邮箱服务器的地址为 `pop.主机名.域名`，而端口的默认值为 110。例如 126 邮箱的 POP3 服务器地址为 `pop.126.com`，端口为默认值 110。如果连接不上，就要查看网站帮助，获取 POP3 服务器的地址和端口。

【实例 14-8】 演示了使用 `poplib` 库来检查指定邮件中的最新 10 封邮件的主题及发信人，代码如下：

```
from poplib import POP3
import re, email, email.header

def decode_email_content(msg_src, names):
    msg = email.message_from_bytes(msg_src)
    result = {}
    for name in names:
        content = msg.get(name)
        info = email.header.decode_header(content)
        if info[0][1]:
            if info[0][1].find('unknown-') == -1:  # 已知编码
                result[name] = info[0][0].decode(info[0][1])
            else:  # 未知编码
                try:
                    result[name] = info[0][0].decode('gbk')
                except:
                    result[name] = info[0][0].decode('utf-8')
        else:
            result[name] = info[0][0]
    return result

if __name__ == "__main__":
    pp = POP3("pop3.163.com")
    pp.user('*****@163.com')
    pp.pass_('*****')
```

```

total,totalnum = pp.stat()
print(total,totalnum)

for i in range(total-10,total):
    hinfo,msgs,octet = pp.top(i+1,0)
    b=b''
    for msg in msgs:
        b += msg+b'\n'
    items = decode_email_content(b,['subject','from'])
    print(items['subject'],'\nFrom:',items['from'])
    print()
pp.close()

```

#访问最近 10 封邮件
#返回的内容为 bytes 类型

【代码说明】程序中定义了一个用 email 包来解码邮件头的函数 decode_email_content；用 POP3 对象的方法连接 POP3 服务器并获取邮箱中的邮件总数，最后获取最近的 10 封邮件的邮件头，传递给函数 decode_email_content 分析并返回邮件的主题和发件人。



注意 代码中的用户名应该写入邮箱的 Email 地址。

【运行效果】用 POP3 对象检查邮件的结果如图 14.11 所示，显示了邮箱中最近接收的 10 封邮件。

```

>>>
411 14922002
认证证书开始申请了! (2月16日至3月11日0点即3月10日24点)
From: 《现代礼仪》课程团队

关于证书的补充通知
From: 《博弈论基础》课程团队

All SUSE Products: 16 New Patches
From: SUSE Patch Notification <patchnotify_noreply@suse.com>

关于证书申请
From: 《数据结构》课程团队

华商基金周刊2015年第7期
From: 华商基金客服中心

All SUSE Products: 5 New Patches
From: SUSE Patch Notification <patchnotify_noreply@suse.com>

感谢学习中国大学MOOC《现代礼仪》课程
From: 中国大学MOOC

All SUSE Products: 5 New Patches
From: SUSE Patch Notification <patchnotify_noreply@suse.com>

All SUSE Products: 6 New Patches
From: SUSE Patch Notification <patchnotify_noreply@suse.com>

蒋文华老师给您拜年啦!
From: 《博弈论基础》课程团队

```

图 14.11 用 POP3 对象检查邮件的结果

14.4.2 用 smtplib 发送邮件

发送邮件一般使用的是 SMTP 协议，使用 Python 的 smtplib 模块可以登录 SMTP 协议发送邮件。要使用 SMTP 协议发送邮件，有两种方式：一种是邮件直接投递，就是说，比如你要发邮件给 aaaa@163.com，那就直接连接 163.com 的邮件服务器，把信投给 aaaa@163.com；另一种是验证过后的发信，它的过程是，比如你要发邮件给 aaaa@163.com，你不是直接投到 163.com，而是通过自己在 sina.com 的另一个邮箱来发，这样就要先连接 sina.com 的 smtp 服务器，然后认证，之后在把要发到 163.com 的信件投到 sina.com 上，sina.com 会帮你把信投递到



163.com。

1. smtplib 模块简介

使用 smtplib 模块的 SMTP 类可以创建一个 SMTP 对象实例。其原型如下：

```
SMTP(host, port, local_hostname)
```

- host 连接的服务器名，可选参数；
- port 服务器端口，可选参数；
- local_hostname 本地主机名，可选参数。

如果在创建 SMTP 对象时没有指定 host 和 port，可以使用 SMTP 对象的 connect 方法连接到服务器，其原型如下：

```
connect(host, port)
```

其参数含义如下：

- host 连接的服务器名，可选参数；
- port 服务器端口，可选参数。

使用 SMTP 对象的 login 方法可以使用用户名和密码登录到 SMTP 服务器。其原型如下：

```
login(user, password)
```

其参数含义如下：

- user 登录服务器的用户名；
- password 登录服务器的密码。

使用 SMTP 对象的 set_debuglevel 方法可以设置调试级别。其原型如下：

```
set_debuglevel(level)
```

其参数 level 是指定的调试级别。

使用 SMTP 对象的 docmd 方法可以向 SMTP 服务器发送命令。其原型如下：

```
docmd(cmd, argstring)
```

其参数含义如下：

- cmd 向 SMTP 服务器发送的命令；
- argstring 命令参数，可选参数。

使用 SMTP 对象的 sendmail 方法可以发送邮件。其原型如下：

```
sendmail(from_addr, to_addrs, msg, mail_options, rcpt_options)
```

其参数含义如下：

- from_addr 发送者的邮件地址；
- to_addrs 邮件发送地址；
- msg 邮件内容；
- mail_options 可选参数，由邮件 ESMTP 操作；
- rcpt_options 可选参数，由 RCPT 操作；

使用 SMTP 对象的 quit 方法可以断开与服务器的连接。

2. 使用 Python 发送邮件

和使用 Python 接受 Email 一样，使用 Python 发送 Email 时，也应该找到所使用 Email 的 SMTP 服务器的地址和端口。对于网易 163 的邮箱，其 SMTP 服务器的地址为 smtp.163.com，端口为默认值 25。为了防止邮件被反垃圾邮件丢弃，这里采用前文中提到的第二种方法，即认证后发送。

【实例 14-9】演示了使用 `smtplib` 库发送邮件，代码如下：

```
import smtplib, email

chst = email.charset.Charset(input_charset='utf-8')
header = ("From: %s\nTo: %s\nSubject: %s\n\n"
          % ("*****@163.com",
            "*****@163.com" ,
            chst.header_encode("Python smtplib 测试!")))
body = "你好!"
email_con = header.encode('utf-8') + body.encode('utf-8')
smtp = smtplib.SMTP("smtp.163.com")
smtp.login("*****@163.com", "*****")
smtp.sendmail("*****@163.com", "*****@163.com", email_con)
smtp.quit()
```

【代码说明】代码中首先应用 `email.charset.Charset()` 对象对邮件头进行编码，之后创建 SMTP 对象，并通过验证的方式给自己发送一封测试邮件。



注意 邮件的主体内容中的中文字符应当使用 `encode()` 进行编码。

【运行效果】收到的邮件在邮箱中显示的效果如图 14.12 所示。



图 14.12 用 SMTP 发送邮件的效果

14.5 用 ftplib 访问 FTP 服务

Python 中的 `ftplib` 模块提供了用于访问 FTP 的函数。使用 `ftplib` 模块可以在 Python 脚本中访问 FTP，完成文件的上传、下载等操作。

14.5.1 ftplib 模块简介

使用 `ftplib` 模块中的 `FTP` 类，可以创建一个 FTP 连接对象。其原型如下：

```
FTP(host, user, passwd, acct)
```

其参数含义如下：

- `host` 要连接的 FTP 服务器，可选参数；
- `user` 登录 FTP 服务器所使用的用户名，可选参数；
- `passwd` 登录 FTP 服务器所使用的密码，可选参数；
- `acct` 可选参数，默认为空。

当创建一个 FTP 连接对象以后，可以使用 `set_debuglevel` 方法设置调试级别。其原型如下：

```
set_debuglevel(level)
```

其参数 `level` 是指调试级别，默认的调试级别为 0。



如果在创建 FTP 连接对象时未使用 HOST 参数,则可以使用 FTP 对象的 connect 方法,其原型如下:

```
connect(host, port)
```

其参数含义如下:

- host 要连接的 FTP 服务器;
- port FTP 服务器的端口, 可选参数。

如果在创建 FTP 对象时未使用用户名和密码,则可以使用 FTP 对象的 login 对象使用用户名和密码登录到 FTP 服务器。其原型及含义如下:

```
login(user, passwd, acct)
```

- user 登录 FTP 服务器所使用的用户名;
- passwd 登录 FTP 服务器所使用的密码;
- acct 可选参数, 默认为空。

使用 FTP 对象的 getwelcome 方法可以获得 FTP 服务器的欢迎信息。使用 FTP 对象的 abort 方法可以中断文件传输。使用 FTP 对象的 sendcmd 和 voidcmd 方法可以向 FTP 服务器发送命令, 这两个方法不同之处在于 voidcmd 方法没有返回值。这两个方法的原型如下:

```
sendcmd(command)  
voidcmd(command)
```

其参数 command 是指向服务器发送的命令字符串。

使用 FTP 对象的 retrbinary 和 retrlines 方法可以从 FTP 服务器下载文件, 不同的是 retrbinary 方法使用二进制形式传输文件, 而 retrlines 方法使用 ASCII 形式传输文件。两者的原型如下:

```
retrbinary(command, callback, maxblocksize, rest)  
retrlines(command, callback)
```

对于 retrbinary, 其参数含义如下:

- command 传输命令, 由“RETR+文件名”组成(之间有空格);
- callback 传输回调函数;
- maxblocksize 设置每次传输的最大字节数, 可选参数;
- rest 设置文件的续传位置, 可选参数。

对于 retrlines, 其参数含义如下:

- command 传输命令;
- callback 传输回调函数。

使用 FTP 对象的 storbinary 和 storlines 方法可以向 FTP 服务器上传文件。这两个方法的不同之处是 storbinary 方法使用二进制形式传输文件, 而 storlines 方法使用 ASCII 形式传输文件。这两个方法的原型如下:

```
storbinary(command, file, blocksize)  
storlines(command, file)
```

对于 storbinary, 其参数含义如下:

- command 传输命令, 由“STOR+文件名”组成(之间有空格);
- file 本地文件句柄;
- blocksize 设置每次读取文件的最大字节数, 可选参数。

对于 storlines, 其参数含义如下:

- command 传输命令;
- file 本地文件句柄。

使用 FTP 对象的 `set_pasv` 方法可以设置传输模式，其原型如下：

```
set_pasv(boolean)
```

其中如果参数 `boolean` 为 `True`，则为被动模式；如果为 `False`，则为主动模式。

使用 FTP 对象的 `dir` 方法可以获取当前目录中的内容列表。

使用 FTP 对象的 `rename` 方法可以修改 FTP 服务器中的文件名。其原型如下：

```
rename(fromname, toname)
```

其参数含义如下：

- `fromname` 原来文件名；
- `tosome` 重命名后的文件名。

使用 FTP 对象的 `delete` 方法可以从 FTP 服务器上删除文件，其原型如下：

```
delete(filename)
```

其中参数 `filename` 是要删除的文件名。

使用 FTP 对象的 `cwd` 方法可以改变当前目录，其原型如下：

```
cwd(pathname)
```

其参数 `pathname` 是要进入目录的路径。

使用 FTP 对象的 `mkd` 方法可以在 FTP 服务器上创建目录，其原型如下：

```
mkd(pathname)
```

其参数 `pathname` 是要创建目录的路径。

使用 FTP 对象的 `pwd` 方法可以获得当前目录。

使用 FTP 对象的 `rmd` 方法可以删除 FTP 服务器上的目录，其原型如下：

```
rmd(dirname)
```

其参数 `dirname` 是要删除的目录。

使用 FTP 对象的 `size` 方法可以获得文件的大小，其原型如下所示：

```
size(filename)
```

其参数 `filename` 是要获取文件大小的文件名。

使用 FTP 对象的 `quit()` 和 `close()` 方法可以关闭与 FTP 服务器的连接。

14.5.2 使用 Python 访问 FTP

Python 的 `ftplib` 模块提供了完整的用于 FTP 协议的函数、方法，使用 `ftplib` 模块可以制作一个简单的类似于 Windows 自带的 FTP 客户端。

【实例 14-10】 演示了使用 `ftplib` 模块创建了一个简单的 FTP 客户端实例，代码如下：

```
# -*- coding:utf-8 -*-
#
from ftplib import FTP                                # 从 ftplib 模块中导入 FTP
bufsize = 1024                                       # 设置缓冲区大小
def Get(filename):                                   # 下载文件
    command = 'RETR ' + filename
    ftp.retrbinary(command, open(filename, 'wb').write, bufsize)
    print('下载成功')
def Put(filename):                                   # 上传文件
    command = 'STOR ' + filename
    filehandler = open(filename, 'rb')
    ftp.storbinary(command, filehandler, bufsize)
    filehandler.close()
    print('上传成功')
```




```

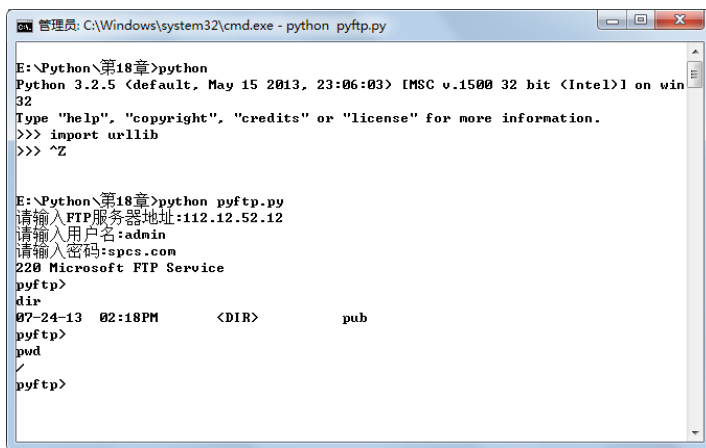
def PWD():                                     # 获取当前目录
    print(ftp.pwd())
def Size(filename):                             # 获取文件大小
    print(ftp.size(filename))
def Help():                                     # 输出帮助
    print('''
=====
        Simple Python FTP
=====
cd          进入文件夹
delete      删除文件
dir         获取当前文件列表
get         下载文件
help        帮助
mkdir       创建文件夹
put         上传文件
pwd         获取当前目录
rename      重命名文件
rmdir       删除文件夹
size        获取文件大小
''')

server = input('请输入 FTP 服务器地址:')      # 获取服务器地址
ftp = FTP(server)                             # 连接到服务器地址
username = input('请输入用户名:')             # 获取用户名
password = input('请输入密码:')               # 获取字典
ftp.login(username,password)                  # 登录 FTP
print(ftp.getwelcome())                       # 获取欢迎信息
actions = {'dir':ftp.dir, 'pwd': PWD, 'cd':ftp.cwd, 'get':Get,
           'put':Put, 'help':Help, 'rmdir': ftp.rmd,
           'mkdir': ftp.mkd, 'delete':ftp.delete,
           'size':Size, 'rename':ftp.rename}

while True:                                   # 命令循环
    print('pyftp>', )                         # 输出提示符
    cmds = input()                            # 获取输入
    cmd = str.split(cmds)                     # 将输入按空格分割
    try:                                       # 异常处理
        if len(cmd) == 1:                     # 判断命令是否有参数
            if str.lower(cmd[0]) == 'quit':    # 如果命令为 quit 则退出循环
                break
            else:
                actions[str.lower(cmd[0])]()    # 调用与命令对应的函数
        elif len(cmd) == 2:                   # 处理命令有一个参数的情况
            actions[str.lower(cmd[0])](cmd[1]) # 调用与命令对应的函数
        elif len(cmd) == 3:                   # 命令有两个参数的情况
            actions[str.lower(cmd[0])](cmd[1],cmd[2])# 调用与命令对应的函数
        else:
            print('输入错误')
    except:
        print('命令出错')
ftp.quit()                                    # 端口连接

```

在命令窗口运行 `pyftp.py` 后，将要求输入 FTP 服务器的地址、用户名、密码，都输入正确后，将完成登录，显示一个“pyftp>”提示符，等待用户输入命令，这时输入“dir”和“pwd”这两个命令后的结果如图 14.13 所示。



```

管理员: C:\Windows\system32\cmd.exe - python pyftp.py

E:\Python\第18章>python
Python 3.2.5 (default, May 15 2013, 23:06:03) [MSC v.1500 32 bit (Intel)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>> import urllib
>>> ^Z

E:\Python\第18章>python pyftp.py
请输入FTP服务器地址:112.12.52.12
请输入用户名:admin
请输入密码:spcs.com
220 Microsoft FTP Service
pyftp>
dir
07-24-13 02:18PM <DIR> pub
pyftp>
pub
pyftp>
/
pyftp>

```

图 14.13 FTP 访问运行演示

注意

要运行 pyftp.py 脚本，需要有一个 FTP 服务器及登录该服务器的用户名和密码，如果没有互联网中的 FTP 服务器，也可自己在本地计算机中通过 IIS 配置一个 FTP 服务器进行测试。

14.6 小结

本章主要介绍了在 Python 语言中进行网络编程的基础知识，如利用 socket 建立客户端与服务器（基于 TCP 协议与 UDP 协议）、利用 socketserver 模块快速建立服务器、用 urllib 与 http 包访问网站、利用 poplib 和 smtplib 检查与发送邮件、用 ftplib 访问 FTP 服务器。

通过本章的学习，你应掌握网络编程的基础知识、建立服务器端与客户端程序、访问 WEB 网站、FTP 站点相关编程知识及编程实现检查与发送邮件。

14.7 本章习题

一、简答题

1. 请说出以下 TCP/IP 协议簇中 FTP、SMTP、HTTP 协议的作用。
2. IP 地址和端口的作用是什么？
3. 端口在 TCP/IP 协议中分为哪几类？你知道 TCP/IP 协议簇中常用协议所使用的默认端口吗？
2. IP 地址和端口的作用是什么？
4. 什么是套接字？它的作用是什么？

二、实验题

1. 编程实现使用通过套接字，分别使用 TCP、UDP 协议实现一个命令行下的聊天程序（模拟两人模式的）。
2. 使用本章所学的库编程实现通过百度获取搜索结果的程序。

第 15 章 线程和进程

现代操作系统大多都是多任务的，可以同时执行多个程序。进程是应用程序正在执行的实体，当程序执行时，也就创建了一个主线程。进程在创建和执行时需要一定的资源，比如内存、文件、I/O 设备等。现代操作系统大多支持多线程和进程。线程是 CPU 使用的基本单元，由主线程来创建，并使用这个进程的资源，因此线程创建成本低而且可以实现并行处理，充分利用 CPU。

Python 提供了对多线程和多进程的支持，在 Python 中可以运用标准库使用多进程和多线程编程。

本章内容包括：

- 线程、进程基础；
- 用 threading 模块进行多线程编程；
- 用 subprocess 模块多进程编程。

15.1 线程



Python3 对多线程支持的是 threading 模块，应用这个模块可以创建多线程程序，并且在多线程间进行同步和通信。在 Python3 中，可以通过以下两种方式来创建线程：通过 threading.Thread 直接在线程中运行函数；通过继承 threading.Thread 类来创建线程。

15.1.1 用 threading.Thread 直接在线程中运行函数

threading.Thread 的基本使用方法如下：

```
Thread(group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None)
```

其中，target 参数就是要运行的函数，args 是传入函数的参数元组。

【实例 15-1】演示了预定义一个函数并以线程方式来运行，代码如下：

```
import threading
def thrfun(x,y):
    for i in range(x,y):
        print(str(i*i)+';')
ta = threading.Thread(target=thrfun,args=(1,6))
tb = threading.Thread(target=thrfun,args=(16,21))
ta.start()
tb.start()
```

【代码说明】本例中先定义了一个 thrfun 函数然后以线程方式来运行，并且每次运行传递不同的参数。

【运行效果】多线程运行的输出信息如图 15.1 所示，两个子线程并行地执行，分别计算出数的平方并输出，很明显，这两个子线程是交替运行的。

```
>>> 1;256;

4;289;

9;324;

16;361;

25;400;
```

图 15.1 多线程运行的输出信息

15.1.2 通过继承 threading.Thread 类来创建线程

这种方法只要重载 threading.Thread 类的 run 方法，然后调用类的 start() 就能够创建线程，并运行 run() 函数中的代码。

【实例 15-2】 预定义了一个函数，并以线程方式来运行，代码如下：

```
import threading
class myThread(threading.Thread):
    def __init__(self, mynum):
        super().__init__()
        self.mynum = mynum

    def run(self):
        for i in range(self.mynum, self.mynum+5):
            print(str(i*i)+';')

ma = myThread(1)
mb = myThread(16)
ma.start()
mb.start()
```

【代码说明】 本例中定义了一个继承了 threading.Thread 类的 myThread 类，创建两个 myThread 类的实例，并用 start() 方法创建和启动线程。

【运行效果】 其运行结果信息与实例 15-1 相同，如图 15.1 所示。



注意

在继承 threading.Thread 类的子类中，如果需要重载 __init__() 方法，必须先调用父类的 __init__() 方法，否则会引发 AttributeError 异常。

15.1.3 线程类 Thread 使用

前文只是继承 Thread 类，重载了其中的两个方法。Thread 类还有几个重要的方法和属性：

【方法】

```
join([timeout])
isAlive()
```

【属性】

```
name
daemon
```

join() 方法的作用是当某个线程或函数执行时需等另一个线程完成操作后才能继续，则应调用另一个线程的 join() 方法；其中的可选参数 timeout 用于指定线程运行的最长时间，isAlive() 方法用于查看线程是否运行。

name 属性是线程设置的线程名；daemon 属性用来设置线程是否随主线程退出而退出，一



般来说，其属性值为 `True` 时不会随主线程退出而退出。

【实例 15-3】演示了 `join()` 函数的基本用法，代码如下：

```
import threading
import time
def thrfun(x,y,thr=None):
    if thr:
        thr.join()
    else:
        time.sleep(2)
    for i in range(x,y):
        print(str(i*i)+';')
ta = threading.Thread(target=thrfun,args=(1,6))
tb = threading.Thread(target=thrfun,args=(16,21,ta))
ta.start()
tb.start()
```

【代码说明】对比实例 15-1 可以看出，对于线程运行的函数 `thrfun()`，若传递的参数中包括一个线程实例，则调用其 `join()` 方法并等待其结束后方才运行；否则，睡眠 2 秒。程序中，`tb` 传入了线程实例 `ta`，因此 `tb` 线程应等待 `ta` 结束后运行。

【运行效果】其运行结果如图 15.2 所示，`tb` 线程的运行输出等到 `ta` 线程输出结束后才输出结果。

```
>>> 1;
4;
9;
16;
25;
36;
256;
289;
324;
361;
400;
```

图 15.2 使用 `join()`



注意

`tb` 线程初始化只能在 `ta` 线程初始化之后，因为没有初始化的线程是不能调用 `join()` 方法的。

【实例 15-4】演示了 `daemon` 属性的作用，代码如下：

```
import threading
import time
class myThread(threading.Thread):
    def __init__(self,mynum):
        super().__init__()
        self.mynum = mynum

    def run(self):
        time.sleep(1)
        for i in range(self.mynum,self.mynum+5):
            print(str(i*i)+';')

def main():
    print('start...')
    ma = myThread(1)
    mb = myThread(16)
    ma.daemon = True
```

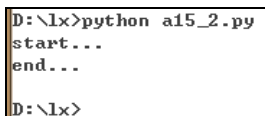
```

mb.daemon = True
ma.start()
mb.start()
print('end...')
if __name__ == "__main__":
    main()

```

【代码说明】main()主函数中初始化了两个线程 ma、mb，可是其 daemon 属性都设置为 True，当 main()函数主线程运行结束时，ma、mb 线程在后台运行，无法输出运行结果。

【运行效果】其运行结果如图 15.3 所示。



```

D:\lx>python a15_2.py
start...
end...
D:\lx>

```

图 15.3 线程无输出



注意

如果在交互式模式下运行该实例，还是会有输出。因为，在交互式模式下的主线程只有在退出 Python 时才中止。

当一个进程拥有了多个线程之后，如果它们各做各的任务还行，可是既然同属于一个进程，它们之间总是具有一定关系的。比如多个线程都要对某个数据进行修改，则可能会出现不可预料的结果。为了保证操作正确，就要对多个线程进行同步。再如，多个线程之间还会需要进行通信。现在就来了解线程的同步和通信。

Python 中可以使用 threading 模块中的对象 Lock 和 RLock（可重入锁）进行简单的线程同步。对于同一时刻只允许一个线程操作的数据对象，可以把操作过程放在 Lock 和 RLock 的 acquire 方法和 release 方法之间。RLock 可以在同一调用链中多次请求而不会锁死，Lock 则会锁死。基本使用方法如下：

```

lock = threading.RLock()    #创建 lock 对象
lock.acquire()               #开始锁定
pass                         #访问或操作多个线程共享的数据
lock.release()               #释放锁

```



注意

在这一对方法中，若前一个调用 n 次，后一个也要调用 n 次，锁才能真正地释放。

【实例 15-5】演示了 RLock 的使用，代码如下：

```

import threading
import time
class myThread(threading.Thread):
    def run(self):
        global x
        lock.acquire()
        for i in range(3):
            x += 10
            time.sleep(1)
            print(x)
        lock.release()
x = 0
lock = threading.RLock()
def main():
    thrs = []
    for item in range(5):
        thrs.append(myThread())

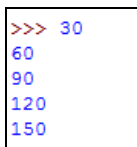
```



```
for item in thrs:
    item.start() if __name__ == "__main__":
main()
```

【代码说明】代码中自定义了一个带锁访问全局变量 `x` 的线程类 `myThread`，在 `main()` 函数中初始化了 5 个线程来修改变量 `x`，但同一时刻只能由一个线程对 `x` 操作。

【运行效果】其运行结果如图 15.4 所示，每次都由一个线程对其加 30 后输出。



```
>>> 30
60
90
120
150
```

图 15.4 加锁访问

你可以修改代码，查看不加锁时的访问结果。

此外，线程间同步还可以用 `threading` 模块中的条件变量、队列等来进行。线程间的通信可以使用 `threading` 模块的 `Event` 对象。`Event` 实例管理着一个内部标志，通过 `set()` 方法会将它设置为 `True`，使用 `clear()` 方法会将它重置为 `False`，`wait([timeout])` 方法会使当前线程阻塞至标志为 `True`。

【实例 15-6】演示了两个线程通过 `Event` 来唤醒对方，模拟人物对话，代码如下：

```
import threading
import time

class myThreada(threading.Thread):
    def run(self):
        evt.wait()
        print(self.name, 'Good morning!')
        evt.clear()
        time.sleep(1)
        evt.set()
        time.sleep(1)
        evt.wait()
        print(self.name, "I'm fine, thank you.")

class myThreadb(threading.Thread):
    def run(self):
        print(self.name, 'Good morning!')
        evt.set()
        time.sleep(1)
        evt.wait()
        print(self.name, 'How are you?')
        evt.clear()
        time.sleep(1)
        evt.set()

evt = threading.Event()
def main():
    John = myThreada()
    John.name = "John"
    Smith = myThreadb()
    Smith.name = 'Smith'
    John.start()
    Smith.start()
if __name__ == "__main__":
    main()
```

【代码说明】代码中自定义了两个线程对象 myThreada、myThreadb，互相以 Event 来进行通信。

【运行效果】两个线程间实现了对象，如图 15.5 所示：

```
>>> Smith :Good morning!
John :Good morning!
Smith :How are you?
John :I'm fine,thank you.
```

图 15.5 模拟对话输出

15.2 进程



Python 虽然支持多线程应用程序的创建，但是 Python 解释器使用了内部的全局解释器锁定(GIL)，在任意指定的时刻只允许单个线程执行，并限制了 Python 程序只能在一个处理器上运行。而现代 CPU 已经以多核为主，但 Python 的多线程程序无法使用。使用 Python 的多进程模块可以将工作分派给不受锁定限制的单独子进程。

Python3 对多进程支持的是 multiprocessing 模块和 subprocess 模块。使用 multiprocessing 模块创建和使用多进程，基本上和 threading 模块的使用方法是—致的。

创建进程使用 multiprocessing.Process 对象来完成，和 threading.Thread 一样，可以用它以进程方式运行函数，也可以通过继承它来并重载 run()方法来创建进程。multiprocessing 模块同样具有 threading 模块中用于同步的 Lock、RLock 及用于通信的 Event，本章不再赘述。

15.2.1 进程基础

subprocess 模块可以用于创建新的进程，并获取它的输入、输出及错误信息。它提供了更高级的接口，可以替换 os.system、os.spawn*、popen 等，subprocess 模块的基本函数如下：

```
call(args, *, stdin=None, stdout=None, stderr=None, shell=False, timeout=None)
    #创建新进程运行程序，输入和输出绑定到父进程，返回新进程退出码
check_call(args, *, stdin=None, stdout=None, stderr=None, shell=False, timeout=None)
    #创建新进程运行程序，输入和输出绑定到父进程，退出码为 0 正常返回，
    #否则，引发 CalledProcessError
getstatusoutput(cmd)
    #创建新进程运行程序，元组形式返回新进程退出码和输出
getoutput(cmd)
    #创建新进程运行程序，返回新进程的输—出（字符串）
check_output(args, *, input=None, stdin=None, stderr=None, shell=False,
    universal_newlines=False, timeout=None)
    #创建新进程运行程序，返回新进程的输—出（bytesarray）
```

参数的基本意义如下：

- stdin、stdout、stderr 用来处理新进程的输入、输出和错误信息；
- shell 是否使用一个中间 shell 来执行(可以使用 shell 相关变量等)；
- input 为命令行提供一个输入信息(字符串)，不能与 stdin 同时用；
- universal_newline 返回值和输入值为字符串而不是 bytes。

【实例 15-7】演示了以上各函数的使用方法及效果，代码如下：

```
import subprocess
print('call() test:',subprocess.call(['python','protest.py']))
print('')
print('check_call() test:',subprocess.check_call(['python','protest.py']))
print('')
```




```
print('getstatusoutput()')
test:', subprocess.getstatusoutput(['python', 'protest.py']))
print('')
print('getoutput() test:', subprocess.getoutput(['python', 'protest.py']))
print('')
print('check_output() test:', subprocess.check_output(['python', 'protest.py']))
```

【代码说明】代码中分别调用这些快捷使用函数，并演示了其输出。子进程运行的是 python 程序，你可以自行新建程序文件，其中输入 `print('Hello World!')` 语句即可。

【运行效果】程序运行效果如图 15.6 所示。

```
D:\>python a15_7.py
Hello World!
call() test: 0

Hello World!
check_call() test: 0

getstatusoutput() test: (0, 'Hello World!')
getoutput() test: Hello World!
check_output() test: b'Hello World!\r\n'
```

图 15.6 subprocess 模块基本函数测试输出



注意

子进程中运行的程序应该在搜索路径中，否则会找不到运行的程序而引发异常。

15.2.2 用 Popen 类创建进程

上节中所述的几个函数，其本质上都是通过 Popen 类来实现的简单版本的进程创建函数。直接使用 Popen 类不仅可以以新进程方式运行命令，还可以对其输入流和输出流等进行更多控制。

Popen 类初始化参数如下：

```
class Popen(args, bufsize=-1, executable=None,
            stdin=None, stdout=None, stderr=None,
            preexec_fn=None, close_fds=True, shell=False,
            cwd=None, env=None, universal_newlines=False,
            startupinfo=None, creationflags=0,
            restore_signals=True, start_new_session=False, pass_fds=())
```

其中的主要参数同上节所述的相同。

Popen 对象具有以下常用方法：

- `poll()` 检查子进程是否结束；
- `wait(timeout=None)` 等待子进程结束；
- `communicate(input=None, timeout=None)` 用于和子进程交互；发送标准输入数据；返回由标准输出和错误输出构成的元组。

其常用属性有：

- `pid` 子进程的 pid；
- `returncode` 子进程的退出码（None 时子进程未退出）。

【实例 15-8】演示了实现使用 Popen 产生子进程的方式执行了 python 源代码，代码如下：

```
import subprocess
prcs = subprocess.Popen(['python', 'protest8.py'],
                        stdout=subprocess.PIPE,
```

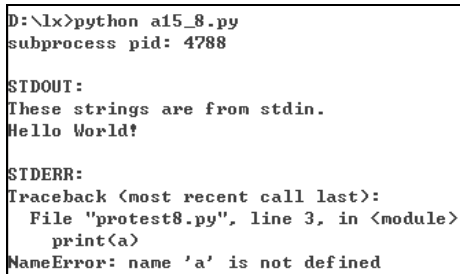
```

        stdin=subprocess.PIPE,
        stderr=subprocess.PIPE,
        universal_newlines=True,
        shell=True)
prcs.communicate('These strings are from stdin.')
print("subprocess pid:",prcs.pid)
print('\nSTDOUT:')
print(str(prcs.communicate()[0]))
print('STDERR:')
print(prcs.communicate()[1])

```

【代码说明】程序中首先用 `Popen` 产生一个子进程，用来执行保存在 `protest8.py` 中的 python 源代码（该源代码要求用户输入一串字符并直接输出和输出另一串写定的字符，以及打印一个未定义的变量 `a`），然后调用 `Popen` 对象的 `communicate()` 方法向子进程传入要输入的字符串，输出子进程的 `pid`，最后分别输出子进程的标准输出和错误信息。

【运行效果】如图 15.7 所示，子进程的 `PID` 为 4788，在输出（`STDOUT`）中输出了通过 `communicate()` 方法向子进程传送的输入 `These strings are from stdin.`，而错误信息中输出了子进程的运行错误提示：`NameError`。



```

D:\>python a15_8.py
subprocess pid: 4788

STDOUT:
These strings are from stdin.
Hello World!

STDERR:
Traceback (most recent call last):
  File "protest8.py", line 3, in <module>
    print(a)
NameError: name 'a' is not defined

```

图 15.7 `Popen` 创建子进程



注意

如果以字符串形式作为子进程的输入，`Popen` 初始化参数 `universal_newlines` 应为 `True`，否则会抛出 `EOFError`。

通过 `communicate()` 方法，可以为多个子进程之间建立“管道线”，即数据从一个进程输入并输出到另一个进程，这有点类似于“击鼓传花”。

【实例 15-8】应用 `communicate()` 方法在子进程间传递信息（类似传递上交花束），代码如下：

```

import subprocess
processes = []
psum = 5
for i in range(psum):
    processes.append(subprocess.Popen(['python', 'protest9.py'],
        stdout=subprocess.PIPE,
        stdin=subprocess.PIPE,
        universal_newlines=True,
        shell=True))

processes[0].communicate('0 bouquet of flowers!')
for before, after in zip(processes[:psum], processes[1:]):
    after.communicate(before.communicate()[0])
print('\nSum of Processes :%d' % psum)
print()
for item in processes:

```



```
print(item.communicate()[0])
```

【代码说明】程序中建立了 psum 个子进程，用来执行文件'protest9.py'中 python 源代码，然后将各个进程的输入和输出连接起来，并输出各子进程的输出信息。文件'protest9.py'中 python 源代码如下：

```
a = input()
a = a.split(' ')
a[0] = str(int(a[0])+1)
print(' '.join(a))
```

【运行效果】如图 15.8 所示：数据每传递过一个进程，都被加 1，由输入时的 0 bouquet of flowers!到进程 5 的 5 bouquet of flowers!。

```
D:\lx>python a15_9.py
Sum of Processes :5
1 bouquet of flowers!
2 bouquet of flowers!
3 bouquet of flowers!
4 bouquet of flowers!
5 bouquet of flowers!
```

图 15.8 子进程管道

15.3 小结

本章主要介绍了线程和进程的基本知识、Python 标准库中的线程和进程相关模块的基本使用。通过学习本章，你应重点掌握 Python 标准库中的 threading 模块和 subprocess 模块，并能使用它们来创建多线程或多进程的 Python 程序，但也要注意，由于受到 Python 语言的全局锁的影响，其多线程只是建立在单 CPU 的基础上运行，并不能充分使用现代计算机中系统中的多 CPU，这也是 Python 的多线程的局限性所在。

15.4 本章习题

一、简答题

1. 什么是线程？在 Python3 中创建线程有哪两种方式？
2. 线程之间的同步可以采用哪几种方式？
3. 什么是进程？在 Python3 中使用进程与使用线程相比有什么优点？

二、实验题

1. 编程实现通过多线程的方式来求 2000~3000 之间的所有素数。
2. 编程通过 Popen 类实现多进程的方式来求 2000~3000 之间的所有素数。

第 16 章 数据库编程

数据库指的是以一定方式存储在一起、能为多个用户共享、具有尽可能小的冗余度、与应用程序彼此独立的数据集合。而平时所说的数据库实际上包含了数据库管理系统（DBMS），数据库管理系统是为管理数据库而设计的软件系统，它一般具有查询、存储、截取、安全保障和备份等功能。

在计算机信息处理中，经常需要保存或处理大量数据，这时就会用数据库来存储这些数据，数据库中的数据按照一定的模型进行组织和存储。Python 提供了对大多数数据库的支持，在 Python 中可以进行连接到数据库、查询数据、添加/删除数据等各种操作。

本章内容包括：

- Python 数据库 API 基础；
- Python 操作 SQLite3；
- Python 操作 MariaDB；
- Python 操作 MongoDB；
- ORM 的框架 mongoengine。

16.1 Python 数据库应用程序接口



为了使 Python 语言对数据库访问具有友好的通用界面，增强 Python 语言使用数据库的可移植性等原因，Python 对于关系数据库的访问接口制定了一个标准，即 Python Database API Specification v 2.0，简称 PEP 249，其参考资料见 <https://www.python.org/dev/peps/pep-0249>。

操作数据的相关第三方模块需要遵从这一规范，但并不是所有第三方模块都严格遵从的。在遵从这一规范的基础上，第三方模块还可以添加一些特别的或有益于用户使用的功能。API 规范中指定了访问数据的两种主要的对象，即连接对象和游标对象。本节介绍这两个对象及其使用方法。

16.1.1 数据库应用程序接口概述

DB-API 2.0 连接对象用来管理数据库连接的对象，它由数据库模块中提供的一个模块级函数 `connect()` 返回的对象。而函数 `connect()` 的参数会因访问的数据库不同而有所不同，但是通常包含如数据源名称、用户名、密码、主机名、数据库名等基础连接信息。

`connect()` 的基本原型如下：

```
connect(dsn, user, password)
```

其中的参数意义：

- `dsn` 数据库服务器主机与数据库名；
- `user` 数据库访问的用户名；
- `password` 数据库访问的密码。

如果连接成功，则返回 `Connection` 对象，它所具有的方法如下：

- `close()` #关闭数据库连接；



- `commit()` #将未完成的事务提交到数据库；
 - `rollback()` #将数据库回滚到未完成事务的开始状态（撤销更改）；
 - `cursor()` #在数据库连接上创建一个 `Cursor` 对象。
- `Cursor` 对象可以用来执行 SQL 语句并获取查询结果，下面将主要讲述 `Cursor` 对象的操作。

16.1.2 数据库游标的使用

`Cursor` 对象执行数据库操作的 SQL 语句或执行存储过程的基本方法及其作用如表 16.1 所示。

表 16.1 `Cursor`对象执行查询的方法及其作用

方 法	作 用
<code>execute(query[,parameters])</code>	在数据库上执行查询或 <code>query</code> 命令（ <code>query</code> 是 SQL 语句字符串）， <code>parameters</code> 是查询字符串中变量值的序列或映射
<code>executemany(query[,paramseq])</code>	多次执行查询命令，将每次查询所需的变量值存储在 <code>paramseq</code> 序列中
<code>callproc(procname[,parameters])</code>	在数据库上调用名为 <code>procnaem</code> 的存储过程，并以 <code>parameters</code> 为参数

`Cursor` 对象获取查询结果数据的基本方法及其作用如表 16.2 所示。

表 16.2 `Cursor`对象获取查询结果集的方法及其作用


方 法	作 用
<code>fetchone()</code>	返回查询数据库后得到的下一行结果集（列表或元组形式）
<code>fetchmany ([size])</code>	返回查询结果行的序列，可选参数 <code>size</code> 代表行数
<code>fetchall()</code>	返回全部剩余的查询结果行的序列
<code>nextset()</code>	跳到下一结果集，准备获取其信息

此外，`Cursor` 对象还有一个关闭指针的方法，即 `close()`方法。
`Cursor` 对象的属性还包括如表 16.3 所示的属性。

表 16.3 游标对象属性及其意义

属 性	意 义
<code>arraysize</code>	为 <code>fetchmany</code> 提供一个默认的整数，表示一次返回结果集行数
<code>description</code>	返回当前结果集的列名信息
<code>rowcount</code>	返回查询结果的行数，-1 则表示没有结果集
<code>nextset()</code>	跳到下一结果集，准备获取其信息

当连接数据库或查询数据库出现错误时，也会引发一些错误，比如 `Interfaceerror`、`DatabaseError`、`DataError`、`InternalError` 等，你应该在程序中处理这些错误。



注意 数据库编程接口具有共性的一面，即大部分数据库操作都是遵循的，但对于具体的数据库也可能有所不同，你应参考其连接库的相关帮助文档。

16.2 Python 操作 SQLite3 数据库

Python 3.x 版本的标准库已经内置了 `sqlite3` 模块，它就是支持 SQLite3 数据库的访问和相关数据库操作，要操作 SQLite3 数据库可以先导入其模块。

16.2.1 SQLite3 数据库简介

SQLite 是一个开源的嵌入式关系数据库，它在 2000 年由 D. Richard Hipp 发布，能减少应用程序管理数据的开销，SQLite 可移植性好、很容易使用、很小、高效而且可靠，作为嵌入式数据库直接在应用程序进程中运行，提供了零配置（zero-configuration）运行模式，并且资源占用非常少。

SQLite3 数据库是将整个数据库（定义、表、索引和数据）都存储在主机端上单一的一个文件中，所以体积很小，一些基本的信息系统都使用它作为基础的数据库。

嵌入式数据库的一大好处就是在你的程序内部不需要网络配置，也不需要管理。因为客户端和服务端在同一进程空间运行。SQLite 的数据库权限只依赖于文件系统，没有用户账户的概念。SQLite 有数据库级锁定，没有网络服务器。它需要的内存和其他开销很小，适合用于嵌入式设备。你需要做的仅仅是把它正确地编译到你的程序中。

程序库实现了多数的 SQL-92 标准，包括事务，就是代表原子性、一致性、隔离性和持久性（ACID）。触发器和多数的复杂查询不进行类型检查，你可以把字符串插入到整数列中。例如，某些用户发现这是使数据库更加有用的创新，特别是与无类型的脚本语言一起使用的时候。其他用户认为这是主要的缺点。

多个进程或线程可以同时访问同一个数据而没有问题。可以同时平行读取同一个数据库。但同一时间只能有一个进程或线程进行数据写入；否则会写入失败并得到一个错误消息（或者会自动重试一段时间，而这重试时间的长短是可以设置的）。

它的主要特点如下。

零配置（Zero Configuration）：不用安装，不用配置，不用启动。

可移植（Portability）：它运行在 Windows、Linux、BSD、Mac OS X 和一些商用 Unix 系统，还可以运行在一些嵌入式系统中。

紧凑（compactness）：SQLite 被设计成轻量级、自包含的。依靠一个头文件、一个 lib 库，你就可以使用关系数据库了。

SQLite3 支持事务，即使在系统崩溃或者突然断电的情况下，SQLite 仍然可以保持原子性、一致性、隔离性和持久性（ACID）。

16.2.2 SQLite3 数据库操作实例

使用 Python 标准库的 sqlite3 包来操作 SQLite3 数据库，依据 DB-API 2.0 规范，主要有以下几个步骤。

1. 导入相关库或模块（sqlite3）。
2. 连接数据库并获取数据连接对象（connect()）。
3. 获取游标对象（con.cursor()）。
4. 使用游标对象的方法（execute()、executemany()、fetchall()等）来操作数据库，对记录进行插入、修改和删除，以及显示相关记录。

sqlite3.connect()连接函数的常用参数有两个：

- database 表示要访问的数据名；
- timeout 表示访问数据的超时设定。

其中，database 就是用字符串的形式指定数据库的名称，如果数据库文件位置不是当前目录，则必须要写出其相对或绝对路径。还可以用“:memory:”表示使用临时放入内存的数据库，而当程序退出时，数据库中的数据也就不存在了。



5. 关闭游标对象和数据库连接 (close())。



注意

数据库操作之后, 应及时调用其 close() 方法关闭数据库连接, 以减轻数据库服务器的压力。

【实例 16-1】演示了运用 sqlite3 模块来操作 SQLite3 数据库的基本方法, 其代码如下:

```
import sqlite3                                #导入标准库 sqlite3
import random                                #导入标准库 random

src = 'abcdefghijklmnopqrstuvwxyz'           #定义随机生成字符串中的所有字符

def get_str(x,y):                             #定义生成字符串函数 (指定长度介于 x,y 间)
    str_sum = random.randint(x,y)            #产生 x,y 间的一个随机整数
    astr = ''
    for i in range(str_sum):
        astr += random.choice(src)           #随机选取 src 中的字符并累加
    return astr

def output():                                 #定义输出数据库表中的所有记录函数
    cur.execute('select * from mytab')         #执行查询
    for sid,name,ps in cur:                   #遍历记录
        print(sid,' ',name,' ',ps)           #输出记录

def output_all():                             #定义输出数据库表中的所有记录函数
    cur.execute('select * from mytab')
    for item in cur.fetchall():               #使用 fetchall() 函数
        print(item)

def get_data_list(n):                         #定义生成记录列表数据的函数
    res = []
    for i in range(n):
        res.append((get_str(2,4),get_str(8,12)))
    return res

if __name__ == '__main__':
    print("建立连接...")
    con = sqlite3.connect(':memory:')         #建立连接 (使用内存中的数据库)
    print("建立游标...")
    cur = con.cursor()                        #获取游标
    print('创建一张表 mytab...')
    cur.execute('create table mytab(id integer primary key autoincrement not
null,name text,passwd text)')                #创建数据库表 mytab
    print('插入一条记录...')
    cur.execute('insert into mytab (name,passwd)values(?,?)',
        (get_str(2,4),get_str(8,12),))      #插入一条记录
    print('显示所有记录...')
    output()                                  #显示所有记录
    print('批量插入多条记录...')
    cur.executemany('insert into mytab (name,passwd)values(?,?)',
        get_data_list(3))                    #插入多条记录
    print("显示所有记录...")
    output_all()                              #显示所有记录
    print('更新一条记录...')
    cur.execute('update mytab set name=? where id=?',('aaa',1))    #更新记录
    print('显示所有记录...')
    output()                                  #显示所有记录
```

```

print('删除一条记录...')
cur.execute('delete from mytab where id=?',(3,))#删除一条记录
print('显示所有记录: ')
output()                                     #显示所有记录
cur.close()                                #关闭游标
con.close()                                #关闭连接

```

【代码说明】代码中首先定义了两个用于生成随机字符串作为数据库记录数据的函数，还有两个分别通过遍历 `cursor`、调用 `cursor` 的 `fetchall()` 方法来获取数据库表中所有记录并输出的函数。然后在主程序中，依次通过建立连接、获取连接的 `cursor`，之后通过 `cursor` 的 `execute()`、`executemany()` 等方法来执行 SQL 语句，以达到插入一条记录、插入多条记录、更新记录和删除记录。最后依次关闭游标和数据库连接。



注意 使用游标的方法来执行 SQL 语句时，其语句中的数据采用占位符(?)的方式。为了防止注入式攻击数据库，一般不采用字符串拼接的方式，而采用参数方式。

【运行效果】如图 16.1 所示，创建了一张表之后，首先插入一条新记录并显示，其次插入多条记录并显示，最后分别更新和删除了一条记录。你可以对比每个操作执行前后显示的数据库表中的数据来了解操作后的数据或记录的变化。

```

>>>
建立连接...
建立游标...
创建一张表mytab...
插入一条记录...
显示所有记录...
1  qyw  xqockarwjwn
批量插入多条记录...
显示所有记录...
(1, 'qyw', 'xqockarwjwn')
(2, 'ey', 'lpmeglry')
(3, 'kpg', 'xngfkvbroyfy')
(4, 'mzys', 'pwrwiflyl')
更新一条记录...
显示所有记录...
1  aaa  xqockarwjwn
2  ey   lpmeglry
3  kpg  xngfkvbroyfy
4  mzys pwrwiflyl
删除一条记录...
显示所有记录:
1  aaa  xqockarwjwn
2  ey   lpmeglry
4  mzys pwrwiflyl

```

图 16.1 Python 操作 SQLite3



注意 更新数据库后应该调用 `connect` 对象的 `commit()` 方法来保存更新结果。

16.3 Python 操作 MariaDB 数据库

MariaDB 数据库是一种开源的数据库，它是 MySQL 数据库的一个分支，因为历史原因，有不少用户担心 MySQL 数据库会闭源，所以 MariaDB 已经发展成替代 MySQL 数据库的主要开源数据库之一。本节介绍的是在 Windows 下运用 MySQL 的官方第三方库来操作 MariaDB 数据库。





16.3.1 MariaDB 数据库简介

MariaDB 数据库管理系统也是一种关系型数据库，主要由开源社区在维护，采用 GPL 授权许可。MariaDB 基于事务的 Maria 存储引擎，替换了 MySQL 的 MyISAM 存储引擎，它使用了 Percona 的 XtraDB、InnoDB 的变体，分支的开发者希望提供访问即将到来的 MySQL 5.4 InnoDB 性能。这个版本还包括了 PrimeBase XT (PBXT) 和 FederatedX 存储引擎。

除了具有 MySQL 数据库系统的特点外，还有自身的一些特性。

1. 兼容性

MariaDB 与 MySQL 在同一分支保持最新的版本，在大多数方面，MariaDB 与 MySQL 几乎一样。所有的命令、接口、库和 APIs 存在于 MySQL，也存在于 MariaDB。切换到 MariaDB 不需要转换数据库。MariaDB 是可以快切换并替代 MySQL 的。此外还可以利用 MariaDB 许多不错的新特性。

2. 速度更快

MariaDB 增加了很多优化及增强功能，在数据的安全复制、索引、字符集转换等方面具有更好的性能。

3. 采用线程池连接查询

使用线程池替代了每个连接，使用查询时对数据的锁定代价降低，在大量的连接下性能有明显提升。

4. 安全性

在 MySQL 的基础上维护自己的一套安全补丁。对于每个 MariaDB 的发布版本，开发人员将合并所有 MySQL 的安全补丁，如果有必要，还会增强它们。

目前，维基百科、Fedora、Slackware Linux、Red Hat 等都已经迁移到 MariaDB 数据库。

16.3.2 建立 MariaDB 数据库操作环境

MariaDB 数据库的安装也很简单，下载其安装版本的 msi 包之后，直接运行安装即可。也可以根据你的计算机操作系统下载其对应版本的压缩包，解压后执行命令安装或在控制台直接应用。本节简单介绍使用 64 位版本的压缩包解压后在控制台直接启动方法。

首先到以下地址下载压缩包并解压到某目录中（此处以 d:\mariadb 为例）。

<https://downloads.mariadb.org/f/mariadb-10.0.17/winx64-packages/mariadb-10.0.17-winx64.zip>
/from/http%3A/mirrors.neusoft.edu.cn/mariadb?serve

(1) 复制数据库目录下的 my-large.ini 到文件 my.ini。

(2) 在命令提示符下使用以下命令以控制台方式启动 MariaDB:

```
bin\mysqld --console
```

启动过程如图 16.2 所示，运行信息显示：bin\mysqld:ready for connections.，表示数据库成功启动，并等待连接。

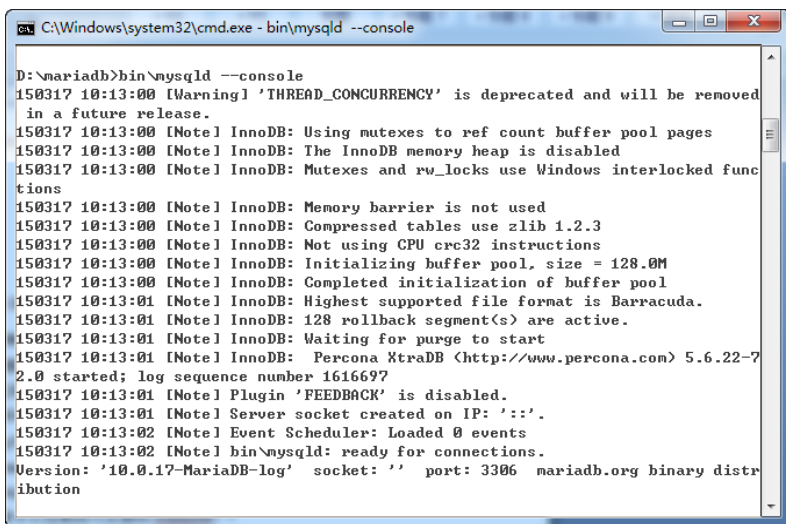


图 16.2 mariaDB 以控制台方式启动

注意

启动后若 Windows 系统弹出如图 16.3 所示的安全警报窗口，请单击“允许访问”，否则将无法通过网络访问数据库。

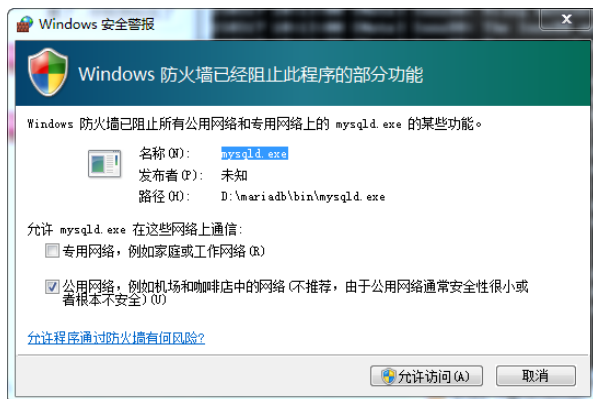


图 16.3 Windows 防火墙警报

这样，数据库就准备就绪了，可以使用 MySQL Connector/Python 来连接数据库并执行相关的操作了。

注意

用以上控制台方式启动 MariaDB 数据库后，默认的数据库 root 用户的密码为空，你可以用以下命令给 root 用户添加一个新密码：

```
bin\mysqladmin -u root password
```

此外，如果希望 MariaDB 数据库能随 Windows 系统一起启动，应该用管理员身份运行以下命令把 MariaDB 数据库安装为服务模式并设置为自动启动或随 Windows 系统自动启动：

```
mysqld install serviceName
```

安装成功的提示如图 16.4 所示。

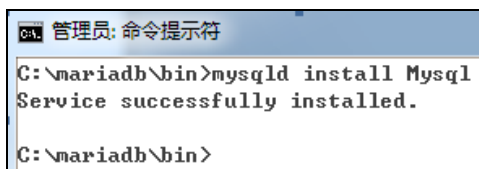


图 16.4 MariaDB 安装命令与提示



注意

此处安装成功后，数据库服务器并不会自动启动，应使用以下命令启动 MariaDB 数据库：

```
net start MariaDB
```

服务启动成功的提示如图 16.5 所示。

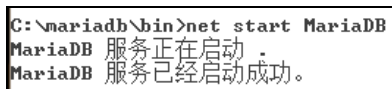


图 16.6 MariaDB 服务启动命令与提示

在运用 Python 的第三方库 MySQL connectorPython 来操作 MariaDB 数据库之前，要先下载和安装这个第三方库，其下载地址为：<http://dev.mysql.com/get/Downloads/Connector-Python/mysql-connector-python-2.0.3-py3.4.msi>。

下载的文件可以直接在 Windows 下运行并启动安装，其具体过程很简单，只要一路单击“next”按钮即可。

16.3.3 MariaDB 数据库操作实例

运用 Python 的第三方库 MySQL connector Python 来操作 MariaDB 数据库和本书前一节中操作 SQLite3 基本相同，其步骤很相似。

mysql-connector-python 模块中的连接函数 connect() 的包路径为 mysql.connector.connect，其函数原型如下：

```
connect(host, port, user, password, database, charset) #仅列出常用参数
```

各参数含义如下：

- host 访问数据库的服务器主机（默认为本机）；
- port 访问数据库的服务端口（默认为 3306）；
- user 访问数据库的用户名；
- password 访问数据库用户名的密码；
- database 访问数据库名称；
- charset 字符编码（默认为 utf8）。

此外，与操作 SQLite3 的 SQL 语句不同的是：SQL 语句中的占位符不是“?”而是“%s”。

【实例 16-2】演示了运用 MySQL 数据的官方提供的第三方库来操作 MariaDB 数据库的基本方法，其代码如下：

```
from mysql import connector #导入第三方库 mysql
import random

src = 'abcdefghijklmnopqrstuvwxyz'
```

```

def get_str(x,y):
    str_sum = random.randint(x,y)
    astr = ''
    for i in range(str_sum):
        astr += random.choice(src)
    return astr

def output():
    cur.execute('select * from mytab')
    for sid,name,ps in cur:
        print(sid,' ',name,' ',ps)

def output_all():
    cur.execute('select * from mytab')
    for item in cur.fetchall():
        print(item)

def get_data_list(n):
    res = []
    for i in range(n):
        res.append((get_str(2,4),get_str(8,12)))
    return res

if __name__ == '__main__':
    print("建立连接...")
    con = connector.connect(user='root',password='',database='test')
    print("建立游标...")
    cur = con.cursor()
    print('创建一张表 mytab...')
    cur.execute('create table mytab(id int primary key auto_increment not null,name
text,passwd text)')
    print('插入一条记录...')
    cur.execute('insert into mytab (name,passwd)values(%s,%s)',
                (get_str(2,4),get_str(8,12)),) #注意占位符为%s
    print('显示所有记录...')
    output()
    print('批量插入多条记录...')
    cur.executemany('insert into mytab (name,passwd)values(%s,%s)',
                    get_data_list(3))
    print("显示所有记录...")
    output_all()
    print('更新一条记录...')
    cur.execute('update mytab set name=%s where id=%s',('aaa',1))
    print('显示所有记录...')
    output()
    print('删除一条记录...')
    cur.execute('delete from mytab where id=%s',(3,))
    print('显示所有记录: ')
    output()

```

【代码说明】此处的代码功能和意义与实例 16-1 基本上是相同的。只是区别在三个地方：

- 导入的包不同（此处为 MySQL）；
- 连接函数参数不同；
- SQL 语句中的占位符不同。

【运行效果】如图 16.7 所示，其输出中除了记录内容（随机产生的）与图 16.1 所示的不同以外，其结构与图 16.1 的是相同的。



```
>>>
建立连接...
建立游标...
创建一张表mytab...
插入一条记录...
显示所有记录...
1  xdoy  giesqwokwk
批量插入多条记录...
显示所有记录...
(1, 'xdoy', 'giesqwokwk')
(2, 'ed', 'dmudmvixz')
(3, 'pqgs', 'utjkhvfvf')
(4, 'nfx', 'xfkreetyq')
更新一条记录...
显示所有记录...
1  aaa  giesqwokwk
2  ed  dmudmvixz
3  pqgs  utjkhvfvf
4  nfx  xfkreetvq
删除一条记录...
显示所有记录:
1  aaa  giesqwokwk
2  ed  dmudmvixz
4  nfx  xfkreetvq
```

图 16.7 操作 MariaDB 数据库实例

16.4 Python 操作 MongoDB 数据库

MongoDB 数据库也是一种开源的跨平台的数据库，与 SQLite3、MariaDB 不同的是，MongoDB 是一种 noSQL（not only SQL）非关系型数据库，MongoDB 数据库的操作方法与它们也是大相径庭的。

16.4.1 MongoDB 数据库简介

MongoDB 是一种强大、灵活、可扩展的数据存储方式，它扩展了关系型数据库的大量有用功能。它的主要特点如下所示。

MongoDB 数据存储没有模式：对于关系型数据库来说，只要建立一个关系，即一张表，那么其中的数据类型基本上就定格了。MongoDB 数据存储基本单元是“文档”（相对于关系型数据库中的记录），每个文档的模式可以不同，不仅数据类型可以不同，其结构也不相同。

MongoDB 具有很强的易扩展性：它所采用的文档数据模型可以自动在服务器之间分割数据，而且其服务器可以集群以平衡服务器的压力。集群的服务器可以自动切换备份的服务器，还可以自动集成和配置新节点。

MongoDB 支持高并发读写：MongoDB 可以通过集群来提高读写性能，甚至可以建立读写共享的集群服务器。

MongoDB 支持海量存储：内置 GridFS，支持大容量的存储，GridFS 是一个出色的分布式文件系统，可以支持海量的数据存储。内置了 GridFS 的 MongoDB，能够满足对大数据集的快速范围查询。

16.4.2 建立 MongoDB 数据库操作环境

由于 MongoDB 数据库是一种开源的跨平台的数据库，所以你可以自由下载和使用，其下载地址为：<http://www.mongodb.org/downloads>。

根据你的计算机系统类型来选择下载相关的压缩包即可，此处以 64 位的 Windows 操作系统为例。

如果你下载的是 MSI 安装包,则需要打开运行安装后才能使用。如果下载的是压缩包,则将下载得到的文件解压缩(本例下载的压缩包并解压到 D:\mongo31 为例),在命令提示符下,就可以通过命令来安装或启动 MongoDB 数据库了。

解压得到的文件夹中的 MongoDB 数据的主要系统文件都在 bin 子目录中,其中 mongod.exe 是数据库启动执行文件, mongo.exe 是数据库的命令行客户端软件,它可以用来查询数据库。

启动数据库的命令 mongod.exe 有很多命令选项,以下是常用的三种命令选项:

- mongod.exe --nojournal --dbpath . #以控制台(无日志)的方式启动服务器(作测试用);
- mongod.exe --install #安装 MongoDB 以服务方式运行;
- mongod.exe --help #显示所有的命令选项。

要建立本小节所需要的测试环境只要使用上述第一种命令选项即可,执行命令的结果如图 16.8 所示,其中最后一行提示说明服务器已经成功启动,使用端口 27017 正在等待客户端的连接。

```
C:\Users\djx>d:
D:\>cd mongo31
D:\mongo31>bin\mongod.exe --nojournal --dbpath .
2015-03-18T19:30:35.935+0800 I CONTROL Hotfix KB2731284 or later update is not
installed, will zero-out data files
2015-03-18T19:30:36.067+0800 I CONTROL [initandlisten] MongoDB starting : pid=3
688 port=27017 dbpath=. 64-bit host=djx-PC
2015-03-18T19:30:36.068+0800 I CONTROL [initandlisten]
2015-03-18T19:30:36.068+0800 I CONTROL [initandlisten] ** NOTE: This is a devel
opment version (3.1.0) of MongoDB.
2015-03-18T19:30:36.068+0800 I CONTROL [initandlisten] ** Not recommended
for production.
2015-03-18T19:30:36.069+0800 I CONTROL [initandlisten]
2015-03-18T19:30:36.069+0800 I CONTROL [initandlisten] targetMinOS: Windows Ser
ver 2003 SP2
2015-03-18T19:30:36.070+0800 I CONTROL [initandlisten] db version v3.1.0
2015-03-18T19:30:36.070+0800 I CONTROL [initandlisten] git version: 7d15cd965cc
b3ad684d8ae4e4f09d5b1e9394552
2015-03-18T19:30:36.071+0800 I CONTROL [initandlisten] build info: windows sys.
getwindowsversion(major=6, minor=1, build=7601, platform=2, service_pack='Servic
e Pack 1') BOOST_LIB_VERSION=1_56
2015-03-18T19:30:36.072+0800 I CONTROL [initandlisten] allocator: system
2015-03-18T19:30:36.073+0800 I CONTROL [initandlisten] options: { storage: { db
Path: ".", journal: { enabled: false } } }
2015-03-18T19:30:36.091+0800 I NETWORK [initandlisten] waiting for connections
on port 27017
```

图 16.8 以控制台方式启动 MongoDB 数据库

当然,有的系统也会弹出如本书图 16.3 所示的警告,你必须单击“允许访问”,使数据库服务器可以通过网络进行访问,而不会被防火墙拦截。

如果需要关闭 MongoDB 数据库服务器,在其启动窗口下按 Ctrl+C 组合键,即可关闭服务器。

下一节介绍 MongoDB 数据库基础及用客户端来操作它。

16.4.3 MongoDB 数据库基础

与关系型数据库相比,作为非关系型的 MongoDB 数据库的有关概念是不同的,如图 16.9 所示。

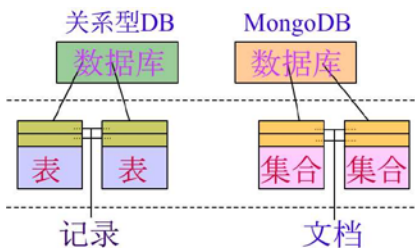


图 16.9 关系型数据库与 MongoDB 概念的对比

一般关系型数据的表和记录在 MongoDB 中的分别叫作集合和文档。记录是表中的一行，文档相当于表的“一行”，但它可以有灵活的嵌套结构，比关系型数据库中的一条记录要复杂得多。

基本的文档可以看作一对或多对的键值对：

```
{ 'name':Bob, age:42 }
```

而复杂的文档可以包含一个数组或嵌套的文档：

```
{ 'name':Bob, 'age':42, 'children':['John','Jim']}           #包含数组的文档
{ 'name':Bob, 'age':42, 'children':{                         #复杂的嵌套文档
  'son':{ 'name':'John','age':12},
  'daughter':{ 'name':'Jim','age':8}
}}
```

并且这些键值对是有序的，比如将第一个示例文档写为：

```
{ age:42,'name':Bob }
```

它们不是同一个文档，但很多情况下键的顺序是无关紧要的，而且有些编程语言，如本书介绍的 Python 也是这样的。

文档中的字符串是区分大小写的，字符串中的字符可以是除了以下情况的所有 UTF-8 字符：

- 键不能含有\0（即空字符）；
- \$和.有特别的意义，只在特定的环境下使用；
- 通常情况下以下画线开头的键是保留的。

要实现对数据库的管理等操作，可以使用上一节所讲的 mongo.exe 客户端来操作 MongoDB，只要直接在命令提示符下运行这个程序，它就会作为客户端默认连接本机的 27012（MongoDB 默认服务端口），如图 16.10 所示，启动成功后出现“>”，就是其客户端的提示符。

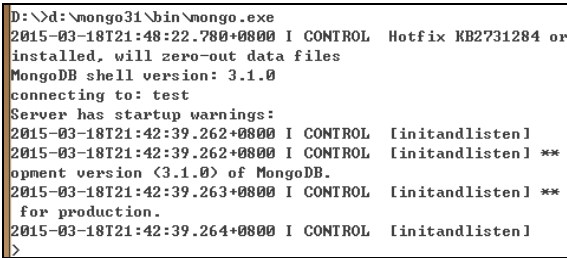


图 16.10 MongoDB 命令行客户端界面

客户端的基本命令可以由 help 命令来获取，如图 16.11 所示。

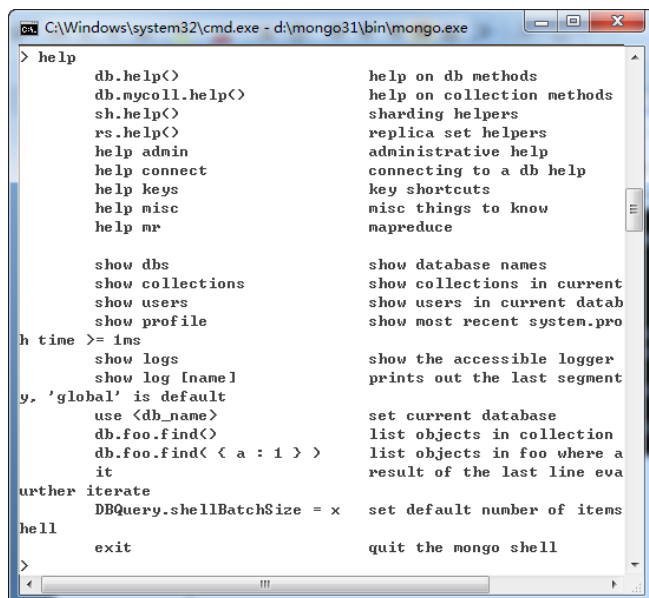


图 16.11 mongo 命令信息

客户端常用的基本命令有：

show dbs	#显示所有数据库名称
show collections	#显示当前数据库中所有集合名称
use dbname	#切换当前数据库
db.foo.find()	#列出 foo 集合中的文档（可加条件过滤）
db.foo.findOne()	#列出 foo 集合中的匹配的文档（可加条件过滤）
db.foo.insert()	#插入一个文档
db.foo.remove()	#删除符合条件的文档
db.update()	#更新文档
db.drop()	#删除集合

以下是一套操作的一些命令实例：

use test	#切换到 test 数据库
db.stu.insert({'name':'liumei','age':15,'grade':2})	#插入一个文档到集合 stu 中
db.stu.find({'grade':2})	#列出集合 stu 中 grade 键值为 2 的文档
db.stu.findOne({'grade':2})	#列出集合 stu 中 grade 键值为 2 的第一个文档
db.stu.update({'name':'hanxuan'},{'\$set':{'age':17}})	#修改集合 stu 中 name 为 hanxuan 的 age 为 17
db.stu.remove({'name':'liming'})	#删除集合 stu 中 name 为 liming 的文档
show dbs	#显示所有数据库名称
show collections	#显示当前数据库中所有集合名称
db.stu.drop()	#删除集合 stu

这些命令在客户端 mongo.exe 上执行其结果如下（带命令提示符“>”的为输入）：

```
> use test                                     #切换到 test 数据库
switched to db test
> db.stu.insert({'name':'liming','age':15,'grade':1}) #在集合 stu 中插入文档
WriteResult({ "nInserted" : 1 })
> db.stu.insert({'name':'zhangsan','age':16,'grade':1})
WriteResult({ "nInserted" : 1 })
> db.stu.insert({'name':'liumei','age':15,'grade':2})
WriteResult({ "nInserted" : 1 })
> db.stu.insert({'name':'hanxuan','age':14,'grade':2})
```




```

WriteResult({ "nInserted" : 1 })
> db.stu.find({'grade':2})                                #查找并返回 grade 为 2 文档
{ "_id" : ObjectId("55098ae657dc226561c045a8"), "name" : "liumei", "age" : 15, "
grade" : 2 }
{ "_id" : ObjectId("55098b0057dc226561c045a9"), "name" : "hanxuan", "age" : 14,
"grade" : 2 }
> db.stu.findOne({'grade':2})                             #查找第一个 grade 为 2 的文档
{
  "_id" : ObjectId("55098ae657dc226561c045a8"),
  "name" : "liumei",
  "age" : 15,
  "grade" : 2
}
> db.stu.update({'name':'hanxuan'},{'$set':{'age':17}}) #更新文档
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.stu.find({'name':'hanxuan'})                         #查找并验证结果
{ "_id" : ObjectId("55098b0057dc226561c045a9"), "name" : "hanxuan", "age" : 17,
"grade" : 2 }
> show dbs                                              #显示所有数据库
local  0.078GB
test   0.078GB
> show collections                                     #显示当前数据库中集合名称
stu
system.indexes
> db.stu.remove({'name':'liming'})                      #删除集合 stu 中指定文档
WriteResult({ "nRemoved" : 1 })
> db.stu.drop()                                         #删除集合 stu
true
>exit                                                  #退出客户端
bye

```

MongoDB 的操作很简练，在使用数据库和集合前根本不需要先建立，而是当执行了插入数据时会自动建立对应的集合和表。而关系型数据库要使用数据和表时需要先用 SQL 语句建立。如上操作实例演示，事先并没有建立 test 数据库，却可以直接用“use test”命令切换；同样没有先创建集合 stu 就可以直接使用 db.stu.insert()来向 stu 集合插入文档。

16.4.4 MongoDB 数据库操作实例

要用 Python 操作 MongoDB 数据库，就需要安装 pymongo 这个第三方库。你可以从以下任一网址下载它，因为后一节的测试环境要求 2.8 版本的 pymongo，建议下载安装 2.8 版本：

<https://pypi.python.org/pypi/pymongo#downloads>

<https://github.com/mongodb/mongo-python-driver>

如果下载的是 exe 的，则可以直接运行安装，如果是压缩包的，可以使用以下命令安装它：

```
python setup.py install
```

安装完成后，可以在交互式环境下使用以下语句来测试安装是否成功：

```
import pymongo
```

pymongo 库来操作 MongoDB 数据库的基本步骤如下：

- (1) 导入 pymongo 库；
- (2) 用 pymongo.MongoClient 类来连接 MongoDB 数据库；
- (3) 用 pymongo.MongoClient 实例选择使用指定的数据库生成数据库对象；
- (4) 用数据库对象生成集合对象，之后就可以用这个对象来操纵数据库了。



注意

以上步骤还是很多的，在实际编程中，以上步骤的 2、3、4 用一句代码即可解决。

其对应的基本代码如下：

```
from pymongo import MongoClient
db = MongoClient()
db_test = db.test          #也可以使用 db_test = ['test']风格的语句
stus = db_test.stus        #也可以使用 stus = ['stu']风格的语句
#以上三行代码可以用一行来完成: stus = MongoClient().test.stu
```

MongoClient()实际是实例化类 MongoClient 的操作，其基本初始化参数包括：

- host 连接的数据库服务器主机；
- port 连接的数据库服务器服务端。

host 默认值就是本地主机，port 的默认值为 MongoDB 数据库服务器默认端口 27017。所以以下的测试中可以不带参数来实例化类 MongoClient。

pymongo 操纵数据库主要是使用集合对象的相关方法完成的，其主要方法如下：

```
insert_one()          #插入一个文档，其参数是一个字典类型的数据
insert_many()         #批量插入多个文档，其参数是一个字典的列表
find_one()           #查询第一个符合条件的文档，其参数是一个字典类型的数据
find()               #查询所有符合条件的文档，其参数是一个字典类型的数据
count()              #计算查询结果集的总数
sort()               #对查询结果集进行排序
```

find()、insert_many()等方法的返回结果也是可迭代获取其值的。



注意

insert_one()、insert_many()和 find_one()是在 3.0 版添加的，而在稳定的 2.8 版只使用 insert()和 fin()方法。

【实例 16-3】演示了使用第三方库 pymongo 操作 MongoDB 数据库的实例，代码如下：

```
from pymongo import MongoClient          #导入 pymongo 数据库
import random

src = 'abcdefghijklmnopqrstuvwxyz'
def get_str(x,y):
    str_sum = random.randint(x,y)
    astr = ''
    for i in range(str_sum):
        astr += random.choice(src)
    return astr

def get_data_list(n):
    res = []
    for i in range(n):
        res.append({'name':get_str(2,4),'passwd':get_str(8,12)})
    return res
if __name__ == '__main__':
    print("建立连接...")
    stus = MongoClient().test.stu        #一条语句实现连接到集合
    print('插入一条记录...')
    stus.insert ({'name':get_str(2,4),'passwd':get_str(8,12)})
    print("显示所有记录...")
    stu = stus.find ()                   #显示刚才插入的一个文档
    print(stu)
    print('批量插入多条记录...')
    stus.insert (get_data_list(3))       #批量插入生成的文档
    print('显示所有记录...')
    for stu in stus.find():              #显示所有的文档
```



```

    print(stu)
print('更新一条记录...')
name = input('请输入记录的 name:')          #输入要更改的文档的 name
stus.update({'name':name}, {'$set':{'name':'aaaa'}}) #更新
print('显示所有记录...')
for stu in stus.find():                       #显示所有文档以验证更改
    print(stu)
print('删除一条记录...')
name = input('请输入记录的 name:')          #输入要删除的文档的 name
stus.remove({'name':name})
print('显示所有记录...')
for stu in stus.find():                       #显示所有文档以验证删除
    print(stu)

```

【代码说明】本例代码的前两个函数的定义和实例 16-1 中是相同的，就是用来产生字符串的。在主程序中，首先连接集合，随后是运用集合对象的方法对集合中的文档进行插入、查找、更新和删除操作。每次数据修改后，显示集合中所有文档，以验证操作结果。

【运行效果】如图 16.12 所示，每一步操作的结果都显示出所有文档。

```

>>>
建立连接...
插入一条记录...
显示所有记录...
{'passwd': 'ottoawayq', '_id': ObjectId('550a608bd87f130bc45f1f18'), 'name': 'fnks'}
批量插入多条记录...
显示所有记录...
{'passwd': 'ottoawayq', '_id': ObjectId('550a608bd87f130bc45f1f18'), 'name': 'fnks'}
{'passwd': 'jmqnqlyzv', '_id': ObjectId('550a608bd87f130bc45f1f19'), 'name': 'xzvv'}
{'passwd': 'fkxejsbyx', '_id': ObjectId('550a608bd87f130bc45f1f1a'), 'name': 'ey'}
{'passwd': 'rcrdeorxgum', '_id': ObjectId('550a608bd87f130bc45f1f1b'), 'name': 'fec'}
更新一条记录...
请输入记录的name:fnks
显示所有记录...
{'passwd': 'ottoawayq', '_id': ObjectId('550a608bd87f130bc45f1f18'), 'name': 'aaaa'}
{'passwd': 'jmqnqlyzv', '_id': ObjectId('550a608bd87f130bc45f1f19'), 'name': 'xzvv'}
{'passwd': 'fkxejsbyx', '_id': ObjectId('550a608bd87f130bc45f1f1a'), 'name': 'ey'}
{'passwd': 'rcrdeorxgum', '_id': ObjectId('550a608bd87f130bc45f1f1b'), 'name': 'fec'}
删除一条记录...
请输入记录的name:aaaa
显示所有记录...
{'passwd': 'jmqnqlyzv', '_id': ObjectId('550a608bd87f130bc45f1f19'), 'name': 'xzvv'}
{'passwd': 'fkxejsbyx', '_id': ObjectId('550a608bd87f130bc45f1f1a'), 'name': 'ey'}
{'passwd': 'rcrdeorxgum', '_id': ObjectId('550a608bd87f130bc45f1f1b'), 'name': 'fec'}

```

图 16.12 pymongo 操作 MongoDB

此外，从图中还可以看出，每个文档被自动加入了一个键值对，即“_id”。它是由服务器的不同进程生成的唯一的一个值，主要由时间、机器标志符、PID、自增序号等部分构成，用来作为每个文档的默认唯一标志。

16.4.5 用对象关系映射（ORM）工具操作 MongoDB 数据库

ORM 是对象关系映射（Object Relational Mapping，或 O/RM）的简称，是一种程序技术，用于实现面向对象编程语言里不同类型系统的数据之间的转换。从效果上说，它其实是创建了一个可在编程语言里使用的“虚拟对象数据库”。

从另外一个角度来看，在面向对象编程语言中使用的是对象，而对象中的数据要保存到数据库中，或数据库中的数据用来构造对象。要从数据库中提取数据并构造对象或将对象数据存入数据库，有很多代码是重复的，而且很多功能完全由自己实现那就是“重复造轮子”。所以就诞生了 ORM，它使得从数据库中提取数据来构造对象或将对象数据保存（持久化）到数据库中实现起来更简单。

通过 ORM 框架来操作数据只需要通过操作类就可以实现一切功能。

较为流行的 ORM 框架有很多,如 Hibernate、Apache OJB、LINQ TO SQL 等。Python 语言的 ORM 框架有 sqlalchemy、mongoKit、mongoengine 等。本小节主要介绍通过 pymongo 访问 MongoDB 数据库的 ORM 框架 mongoengine。

mongoengine 的使用基本步骤如下:

- (1) 导入 mongoengine 库;
- (2) 连接 MongoDB 数据库;
- (3) 定义对象——数据类,会自动在数据库中建立一个对应的集合;
- (4) 在程序或项目中直接调用对象及其方法来完成数据的持久化。

其对应的示例代码如下:

```
from mongoengine import *
connect('test')
class User(Document):
    name = StringField()
    age = IntField()
    passwd = StringField()
```

自定义对象实例的持久化相关方法有:

save()	#保存对象数据到数据库中
update()	#更新对象数据到数据库中
update_one()	#根据对象数据更新一个匹配的文档
delete()	#删除对象数据(从数据库中)
objects()	#查询数据库中符合条件的文档

【实例 16-4】演示了 mongoengine 的使用实例,其代码如下:

```
import random
from mongoengine import *                                #导入 mongoengine 库

connect('test')                                          #连接数据库 test

class Stu(Document):                                    #定义 ORM 框架类
    sid = SequenceField()                                #定义序号属性
    name = StringField()                                #定义字符串类型属性
    passwd = StringField()                               #定义字符串类型属性

    def introduce(self):                                  #定义显示自己的方法
        print('序号:',self.sid,end=" ")
        print('姓名:',self.name,end=' ')
        print('密码:',self.passwd)

    def set_pw(self,pw):                                  #定义修改密码的方法
        if pw:
            self.passwd = pw
            self.save()

src = 'abcdefghijklmnopqrstuvwxyz'

def get_str(x,y):
    str_sum = random.randint(x,y)
    astr = ''
    for i in range(str_sum):
        astr += random.choice(src)
    return astr

if __name__ == '__main__':
```



```

print('插入一个文档:')
stu = Stu(name='lilei',passwd='123123')    #创建一个类（对应一个文档）
stu.save()                                #持久化类（保存文档）

stu = Stu.objects(name='lilei').first()     #查询出数据并初始化类

if stu:
    stu.introduce()                        #显示类（文档）信息
print('插入多个文档')
for i in range(3):
    Stu(name=get_str(2,4),passwd=get_str(6,8)).save()    #插入三个文档

stus = Stu.objects()                      #查询所有文档
for stu in stus:
    stu.introduce()                        #遍历文档并逐个显示

print('修改一个文档')
stu = Stu.objects(name='lilei').first()    #查询某个文档（自动化构建为类）
if stu:
    stu.name='aaaa'                       #修改实例属性
    stu.save()                             #持久化入数据库
    stu.set_pw('bbbbbbbb')                #调用类的业务方法，修改 passwd
    stu.introduce()

print('删除一个文档')
stu = Stu.objects(name='aaaa').first()     #查询获取一个文档
stu.delete()                              #删除一个文档

stus = Stu.objects()                      #查询所有文档
for stu in stus:
    stu.introduce()                        #遍历文档并逐个显示

```

【代码说明】本示例代码仍然是修改前例代码，实现相同的功能。在导入 `mongoengine` 库和连接 `MongoDB` 数据库之后，定义了一个继承 `Document` 类的类 `Stu`。在主程序中通过创建类的实例，并调用其 `save()` 方法将类持久化到数据库；通过 `Stu` 的类方法 `objects()` 来查询数据库并映射为类 `Stu` 的实例，并调用其自定义方法 `introduce()` 来显示载入的信息。然后通过查询文档自动化载入和实例的类也可以修改类的属性，并调用其 `save()` 方法持久化存入数据库，还可以调用类的自定义方法 `set_pw()` 来修改数据并存入数据库。最后是通过调用类的 `delete()` 方法从数据库中删除一个文档。

【运行效果】每个操作的结果信息如图 16.13 所示。

```

>>>
插入一个文档：
序号：1 姓名：lilei 密码：123123
插入多个文档
序号：1 姓名：lilei 密码：123123
序号：2 姓名：yk 密码：npmcjyq
序号：3 姓名：twi 密码：zslefave
序号：4 姓名：sr 密码：xjxlohub
修改一个文档
序号：1 姓名：aaaa 密码：bbbbbbbb
删除一个文档
序号：2 姓名：yk 密码：npmcjyq
序号：3 姓名：twi 密码：zslefave
序号：4 姓名：sr 密码：xjxlohub

```

图 16.13 mongoengin 操作 MongoDB

16.5 小结

本章首先介绍了 Python 语言的数据库基础——DB-API 2.0 中的连接对象和游标对象，然后介绍通过它们来访问或操作内嵌的 SQLite3 数据库、MariaDB 数据库等关系型数据库，并各举出一个包含连接数据库、插入、修改、删除数据库中数据的一个实例，此外还介绍了 MariaDB 数据环境的建立。最后重点介绍了 Python 操作非关系型的 MongoDB 数据库的基本方法，包括建立其数据库服务器测试环境、MongoDB 数据库基础知识及客户端程序的使用、用 pymongo 操纵数据和用 ORM 框架 mongoengine 来操纵数据库。通过学习本章的内容，你应该掌握 Python 的基本数据库应用程序接口及其使用方法，并能使用它们连接和操纵数据库中的信息。

16.6 本章习题

一、简答题

1. 什么是 Python 的数据库应用程序接口？其主要包括哪些数据库操作的对象？
2. 使用 Python 的数据库应用程序接口连接和操作数据库的基本步骤是什么？
3. MongoDB 是一种什么类型的数据库？它有什么优点？
4. 什么是对象关系映射（ORM）工具？

二、实验题

1. 使用 SQLite3 作为数据库编程实现一个简版的通讯录，通讯录的字段主要有姓名、电话、手机号、电子邮箱、QQ，可以实现存储和查找。
2. 使用 MariaDB 作为数据库编程实现一个通讯录，通讯录的字段主要有姓名、电话、手机号、电子邮箱、QQ，可以实现存储和查找，并且可以实现通讯录的分组。
3. 使用 MongoDB 和 ODM 工具 mongoengine 来实现一个通讯录，通讯录的字段主要有姓名、电话、手机号、电子邮箱、QQ，可以实现存储和查找，并且可以实现通讯录的分组。

第 17 章 Web 网站编程

Web 编程是程序设计应用之一，随着动态网站不断发展，Web 编程已经成为程序设计的重要应用领域。Web 编程主要有 ASP.NET、PHP、Java 等编程语言，Python 语言也可以像其他语言一样应用于 Web 服务。

为了减轻 Web 编程的工作量，在进行 Web 编程时，总是要使用 Web 编程框架。这样，程序员只需要关注 Web 页面的实现。Python 语言的 Web 框架的种类也很多，比较流行的有 Django、Plone、TurboGears、Pyramid、Web.py、Zope2、Flask、tornado、bottle、uliWeb 等，有的功能大而全，有的属于功能较少的微框架。微框架的优点是小巧而灵活，但有些功能没有提供，要由程序自己实现或者使用第三方扩展。本章主要介绍 Flask、Tornado 两个微框架的应用。

本章内容包括：

- Flask 框架安装；
- Flask 框架及其应用；
- Tornado 框架安装；
- Tornado 框架及其应用；
- Tornado 框架开发网站实例。

17.1 Web 网站编程概述



Web 是一个由许多互相链接的超文本组成的系统，通过互联网访问。这些超文本内容通过超文本传输协议（Hypertext Transfer Protocol）传送给用户，而后者通过单击链接来获得相关内容，其基本过程如图 17.1 所示。



图 17.1 Web 工作原理

客户端使用浏览器（Browse）通过互联网和 HTTP 协议向服务器发出请求，服务器收到请求后，如果包含相关资源，则会以 HTTP 协议向客户端回应其请求的相关内容。这种工作模式通常被简称为 B/S（Browse/Server）模式，也是较为流行的软件工作模式之一。

Web 编程一般分为前端和后端编程：前端编程就是使用 HTML 语言写出网页框架及有关内容，使用 Javascript 写在客户端运行的与用户或服务器端交互的程序；后端编程的主要任务是完成网站与客户端相关信息的交换与处理，也包括将数据存入数据库和从数据库中查询出数据。

本书所介绍的 Web 编程主要是指 Web 后端编程，即通过服务器端程序实现业务逻辑，完成数据处理的相关功能，比如接受用户数据、对用户数据进行校验并保存至数据库中，再如从数据库中查询出相关数据，用来校验或将数据发送给用户。

17.2 Flask Web 框架及其应用

Flask 是一个使用 Python 编写的轻量级 Web 应用框架，它的安装依赖 Werkzeug WSGI 工具箱和 Jinja2 模板引擎，并且使用 BSD 授权。

17.2.1 Flask Web 框架简介

Flask 也被称为“microframework”，因为它使用简单的核心，用 extension 增加其他功能。Flask 没有默认使用的数据库、窗体验证工具。然而，Flask 保留了扩展的弹性，可以用 Flask-extension 加入以下功能：ORM、窗体验证工具、文件上传、各种开放式身份验证技术等。

作为 Web 框架的 Flask，其主要特点如下：

- 核心简单而且可扩展；
- 支持 secure cookies (client side sessions)；
- Unicode based；
- 内置开发用服务器和 debugger；
- 集成单元测试 (unit testing)；
- 100% WSGI 1.0 兼容；
- 使用 Jinja2 模板引擎；
- 可用 Extensions 增加其他功能。

总之，Flask 可以变成你想要的任何东西，一切恰到好处，由你做主。Flask 通过扩展为你的应用添加这些功能，就如同这些功能是 Flask 原生的一样。Flask 可能是“微小”的，但它已经为满足用户的各种生产需要做出了充足的准备。

17.2.2 Flask Web 框架初识

Flask 框架并不是 Python 语言的标准库，所以要使用它，必须先进行安装。使用 pip 命令来安装最为简单，因为它会自动帮你安装其依赖的第三方库，在命令提示符下使用以下安装命令：

```
pip install flask
```

安装完成后，在交互式环境下使用 import Flask 语句，如果没有错误提示，则说明安装成功。

如果你要通过下载手动安装，必须先下载安装 Flask 依赖的两个外部库，即 Werkzeug 和 Jinja2，分别解压后进入对应的目录，在命令提示符下使用 python setup.py install 来安装它们。它们的下载地址分别为：

<https://github.com/mitsuhiko/jinja2/archive/master.zip>

<https://github.com/mitsuhiko/werkzeug/archive/master.zip>

然后再下载并用相同的命令来安装 Flask，其下载地址为：

<http://pypi.python.org/packages/source/F/Flask/Flask-0.10.1.tar.gz>

就像学习一门编程语言一样，要了解 Flask 框架，最好的方法是首先了解其基本应用的建立方法。下面通过一个最简单的 Web 实例来说明 Flask 框架的基本使用方式。

【实例 17-1】演示了使用 Flask 框架建立的一个最简单的 Web 程序，代码如下：

```
import flask                                #导入 flask

app = flask.Flask(__name__)                 #实例化主类 Flask
```




```
@app.route('/')                                #装饰器（实现 URL 地址）
def hello():                                    #定义业务函数
    return '你好，我是 Flask!'                  #返回字符串

if __name__ == '__main__':
    app.run()                                    #运行程序
```

【代码说明】从以上代码可以看出，写一个 Web 服务程序一共还不到 10 行代码。导入 Flask 并实例化其主类；然后自定义了只返回一串字符的最简单的函数 `hello()`，并用装饰器 `@app.route('/')` 来装饰它，将 URL 和这个函数联系起来，使得服务器收到对应的 URL 请求时，调用这个函数，返回这个函数生产的数据。

【运行效果】如图 17.2 所示，这一行提示表示 Web 服务器已经正常启动运行了，它的默认服务器端口为 5000，IP 地址为 127.0.0.1（代表本机，访问时也可以使用 `localhost` 表示），通过 `Ctrl+C` 组合键退出服务器。

```
>>>
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

图 17.2 基本 Web 服务器程序

如果要验证这个 Web 服务程序是正常运行的，可以打开浏览器，在地址栏中输入如图 17.2 所示的网址，就可以访问这个最简单的服务器了。如图 17.3 所示，访问得到的页面内容为“你好，我是 Flask！”，这正是本例代码中的函数 `hello()` 返回的字符串。

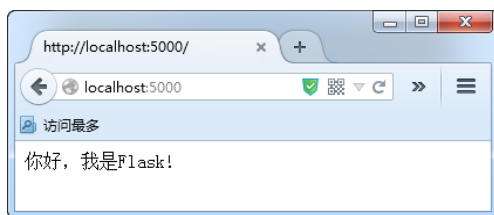


图 17.3 访问简单服务器页面

当浏览器访问发出请求被服务器收到后，服务器还会显示出相关信息，如图 17.4 所示，表示访问该服务器的客户端地址、访问的时间、请求的方法以及表示访问结果的状态码。

```
>>>
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [20/Mar/2015 21:29:12] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [20/Mar/2015 21:29:13] "GET /favicon.ico HTTP/1.1" 404 -
127.0.0.1 - - [20/Mar/2015 21:29:13] "GET /favicon.ico HTTP/1.1" 404 -
```

图 17.4 收到客户端请求显示的信息

上例中主程序里 Flask 实例的 `run()` 方法在调用时全部使用的是参数的默认值，这个方法的参数主要有：

- `host` 服务的 IP 地址，默认为 `None`；
- `port` 服务的端口，默认为 `None`；
- `debug` 是否开启调试模式，默认为 `None`。



注意

此处是直接在 IDLE 中运行的，如果要更好地调试，请在命令提示符下使用“`python filename.py`”来运行和调试服务器。

17.2.3 URL 装饰器与 URL 参数传递

通过实例 17-1 代码可以看出, URL 装饰器是 Flask 实例的 `route()` 方法, 它可以将一个普通函数与特定的 URL 关联起来, 让服务器收到这个 URL 请求时将调用这个函数以返回相应内容。

一个函数可以由多个 URL 装饰器来装饰, 实现多个 URL 请求由一个函数产生的内容回应。对上一节实例稍加修改就可以实现将不同的 URL 映射到同一个函数上。

【实例 17-2】 演示了多个 URL 装饰器同时装饰一个函数的实例, 代码如下:

```
import flask                                #导入 flask

app = flask.Flask(__name__)                 #实例化主类 Flask

@app.route('/')                              #装饰器 (实现 URL 地址)
@app.route('/hello')
def helo():                                #定义业务函数
    return '你好, 我是 Flask!'             #返回字符串

if __name__ == '__main__':
    app.run()                               #运行程序
```

【代码说明】 本例代码只比实例 17-1 多了一行, 即装饰器语句, 实现了多个 URL 映射到一个函数上。

【运行效果】 如图 17.5 所示, 通过图中上方的服务器信息可以看出, 两次 URL 请求分别为 “/” 和 “/hello”, 在服务器端这两个 URL 请求被映射到同一个函数 `helo()`, 所以从图 17.5 下方的两个浏览器窗口可以看出它们显示的内容完全一致。

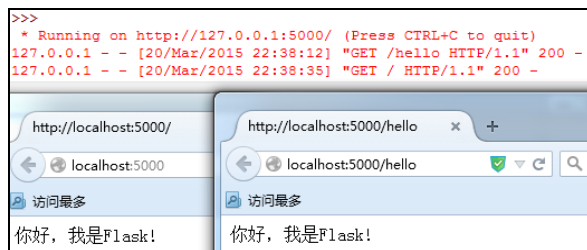


图 17.5 多个 URL 装饰器装饰同一个函数

Flask 中的 URL 装饰器有另一个参数, 即请求的方法类型。众所周知, HTTP 请求最常用的两种方法是 “GET” 和 “POST”, 而 URL 装饰器的默认方法为 “GET”。那么要让同一 URL 的两种请求的方法都映射在同一个函数上, 就必须使用这个参数。



注意

GET 请求一般是直接在浏览器地址栏中输入 URL 或单击链接产生的请求方式; POST 请求一般由提交表单时根据指定的请求方法 POST, 而产生的对 Web 服务器的请求。

【实例 17-3】 演示了使用这个参数的 URL 装饰器的实例, 代码如下:

```
import flask                                #导入 flask
html_txt = ""                               #HTML 页面字符串 (GET 请求的页面代码)
<!DOCTYPE html>
<html>
    <body>
        <h2>收到 GET 请求</h2>
```



```

<form method='post'>                                #指明请求方法为 POST
<input type='submit' value='发送 POST 请求' />
</form>
</body>
</html>
"""
app = flask.Flask(__name__)                            #实例化主类 Flask
#以下装饰器说明 URL 为 '/hello' 的请求，不论是 'GET' 还是 'POST' 方法，都映射到该函数
@app.route('/hello',methods=['GET','POST'])
def hello():                                           #定义业务函数
    if flask.request.method == 'GET':                 #判断收到的请求是否为 GET
        return html_txt                             #返回 GET 请求的页面内容
    else:                                             #否则收到的是 POST 请求
        return '收到 POST 请求，我是 Flask! '       #返回 POST 请求的页面内容
if __name__ == '__main__':
    app.run()                                         #运行程序

```

【代码说明】代码中预先定义了 GET 请求要返回的页面内容字符串 html_txt，在业务函数 hello() 的装饰器中提供了 methods 参数为“GET”“POST”字符串列表，表示 URL 为‘/hello’的请求，不论是‘GET’还是‘POST’方法，都映射到该函数。在业务函数 hello() 内部使用 flask.request.method 来判断收到的请求方法是“GET”还是“POST”方法，然后分别返回不同的内容。

【运行效果】如图 17.6 所示，通过图中第三行、第四行的服务器信息可以看出，两次 URL 请求分别为“GET /hello”和“POST /hello”。在客户端的浏览中，GET 请求的结果页面如图 17.7 所示，POST 请求的结果页面如图 17.8 所示。

```

D:\lx\17>a17_3.py
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [21/Mar/2015 08:08:53] "GET /hello HTTP/1.1" 200 -
127.0.0.1 - - [21/Mar/2015 08:08:57] "POST /hello HTTP/1.1" 200 -

```

图 17.6 服务器端显示请求方法

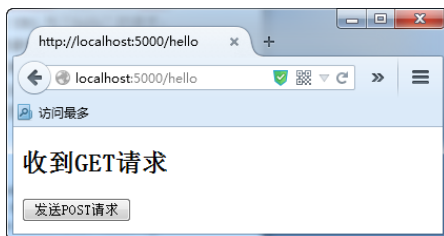


图 17.7 GET 请求的结果

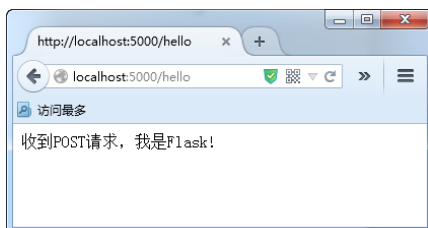


图 17.8 POST 请求的结果

在一般情况下，通过浏览器来传参数相关数据或参数都是通过 GET 或 POST 请求中包含参数来实现的。实际上通过 URL 也是可以传递参数的。通过 URL 传递参数的方法是直接将数

据放入 URL 中，然后在服务器端获取。

Flask 中获取 URL 参数要在 URL 装饰器和业务函数中分别进行定义或处理，URL 变量规则有以下两种形式（URL 装饰器中的 URL 字符串写法）：

```
/hello/<name>          #将形如“/hello/wangfei”URL 中的参数“wangfei”赋值给 name 变量
/hello/<int: id>        #获取“/hello/3”URL 中的参数“3”并自动转换为整数 3 给 id 变量
```

要获取和处理 URL 中传来的参数，就要在对应的业务函数的参数列表中列出变量名，其形式如下：

```
@app.route("/hello/<name>")
def get_url_param(name):
    pass
```

这样，在业务函数 get_url_param() 中就可以引用这个变量值，并进一步使用 URL 中传来的参数。

【实例 17-4】演示了使用这个参数的 URL 装饰器的实例，代码如下：

```
import flask                                #导入 flask

app = flask.Flask(__name__)                 #实例化主类 Flask

@app.route('/hello/<name>')
def helo(name):                             #定义业务函数,列出参数 name
    return "你好," + name + '!'            #返回请求的页面内容
if __name__ == '__main__':
    app.run()                               #运行程序
```

【代码说明】代码中使用了带 URL 参数的 URL 装饰器，会将匹配的参数赋予 name 变量，在业务函数 helo() 中对应地列出变量名 name，用来接收参数值，在业务函数中就可以使用传来的值了。

【运行效果】如图 17.9 所示，URL 请求中包含了参数“李明”，在服务器端的业务函数中引用它并返回给浏览器请求内容中也包含了这个参数值。

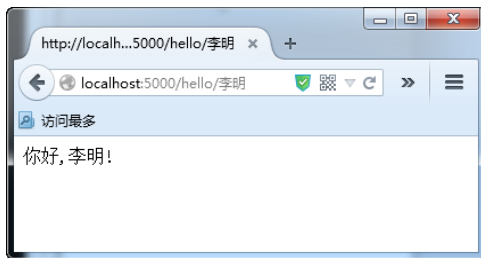


图 17.9 通过 URL 传递参数

17.2.4 GET 与 POST 请求的参数传递

实际上，客户端与服务器传递参数的主要方式还是使用 GET 或 POST 参数传递，Flask 也提供了简单的参数获取方法。

获取 GET 请求参数的基本方式是调用 flask.request.args 对象的 get() 方法，形式如下：

```
flask.request.args.get(name)                #name 为参数名称
```

如果是 POST 请求的参数，需要使用 flask.request 的 form 字典对象来获取，调用形式如下：

```
flask.request.form['name']                  #name 为参数名称
```



当用这种方法获取的参数名不存在时会导致 400 错误请求页面。因此要使用这种方法来获取参数时，可以使用短 if 语句或逻辑运算形式：

```
name = flask.request.form['name'] if 'name' in flask.request.form else None
name = 'name' in flask.request.form and flask.request.form['name']
```

第一种形式是根据 'name' in flask.request.form 判断结果来确定值，即 form 字典中有 'name' 键时，就获取它，否则结果为 None。

第二种形式是采用 Python 语言的逻辑运算的特点实现的，即 and 运算符结算结果始终是决定其整个运算结果的表达式，在这里表现为 'name' in flask.request.form 为 False 时，无论 and 后表达式值为何种情况，结果为 False，所以只返回 False。如果 'name' in flask.request.form 为 True，整个逻辑运算结果就取决于第二个表达式的值，所以会返回 flask.request.form['name'] 的值。



注意 这种方式也能获取 GET 请求参数。

【实例 17-5】演示了获取 POST 请求参数的一个实例，代码如下：

```
# -*- encoding:utf-8 -*-
import flask

html_txt = """                                #定义 GET 请求时页面内容
<!DOCTYPE html>
<html>
    <body>
        <h2>收到 GET 请求</h2>
        <form method='post'>
            <input type='text' name='name' placeholder='请输入你的姓名' />
            <input type='submit' value='发送 POST 请求' />
        </form>
    </body>
</html>
"""

app = flask.Flask(__name__)

@app.route('/hello', methods=['GET', 'POST'])
def hello():
    if flask.request.method == 'GET':
        return html_txt
    else:
        #获取参数 name 的值
        name = 'name' in flask.request.form and flask.request.form['name']
        if name:
            return '你是: ' + name + '!'
        else:
            return '你没有输入姓名! '

if __name__ == '__main__':
    app.run(debug=True)
```

【代码说明】代码和实例 17-3 代码是相似的。在页面代码字符串中添加输入姓名的表单，在业务函数 hello() 中，添加了获取 POST 参数的语句并根据用户的输入返回页面的内容。另外，此处的主程序 run() 函数中使用了参数 debug=True，这样，每次修改保存代码后，服务器会自动重新启动，并且当业务函数引发错误时，会在页面中显示 debug 信息用来调试。

【运行效果】GET 请求的页面如图 17.10 所示，当用户输入“王伟”时，POST 请求的页面内容如图 17.11 所示，当用户没有输入内容时，POST 请求的页面内容如图 17.12 所示。

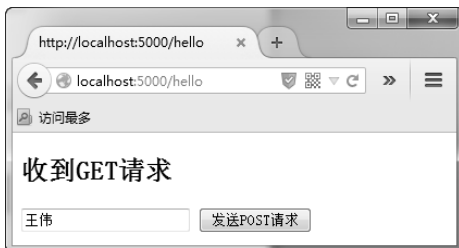


图 17.10 GET 请求页面

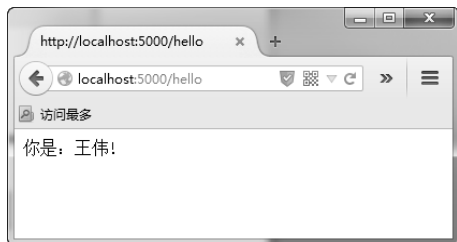


图 17.11 有参数 POST 请求页面

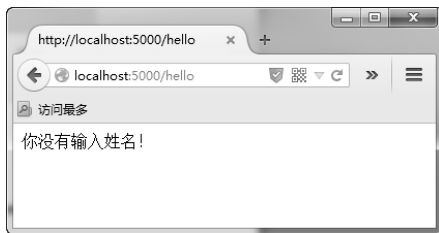


图 17.12 无参数 POST 请求页面

17.2.5 使用 cookie 与 session 跟踪客户

HTTP 协议是一种无状态的协议，当服务器收到客户端请求时，无法知晓请求的客户端是谁。因此，产生了 cookie 与 session 用来存储交互状态。cookie 是运用客户端存储交互状态的一种方式，而 session 则主要是在服务器端存储交互状态的一种方式。

Flask 框架也提供了这两种常用的交互状态存储方式，但是 session 与有些框架是不同的。Flask 中的 session 是以密钥签名加密的 cookie，即用户可以查看你的 cookie，但是如果没有密钥就无法修改它，而且也是保存在客户端的。

获取 cookie 可以使用：

```
flask.request.cookies.get('name')
```

设置 cookie 则需要使用 make_response 对象：

```
resp = make_response(content)           #content 返回页面内容
resp.set_cookie('username', 'the username') #设置名为 username 的 cookie
```

【实例 17-6】演示了一个使用 cookie 跟踪用户的实例，代码如下：





```
# -*- encoding:utf-8 -*-
import flask

html_txt = """                                #定义设置 cookie 页面内容
<!DOCTYPE html>
<html>
    <body>
        <h2>收到 GET 请求</h2>
        <a href='/get_info'>获取 cookie 信息</a>
    </body>
</html>
"""

app = flask.Flask(__name__)

@app.route('/set_info/<name>')                #从 URL 中获取参数 URL 装饰器
def set_cks(name):
    name = name if name else 'anonymous'
    resp = flask.make_response(html_txt)      #构造响应对象
    resp.set_cookie('name', name)            #设置 cookie
    return resp

@app.route('/get_info')
def get_cks():
    name = flask.request.cookies.get('name') #获取 cookie 信息
    return '获取的 cookie 信息是:' + name    #返回带 cookie 信息的页面内容

if __name__ == '__main__':
    app.run(debug=True)
```

【代码说明】代码中定义了两个业务函数，第一个业务函数用于从 URL 中获取参数并将其保存入 cookie 中；第二个业务函数是从 cookie 中读取数据并显示在页面中。

【运行效果】浏览器使用 URL “localhost:5000/set_info/李三” 来设置名为 name 的 cookie 信息，如图 17.14 所示，当单击“获取 cookie 信息”链接进入 “/get_info” 时，页面中将显示 cookie 中保存的 name 名称的信息，如图 17.15 所示。

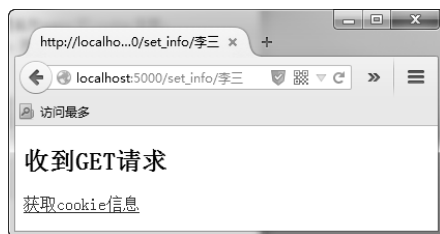


图 17.14 设置 cookie

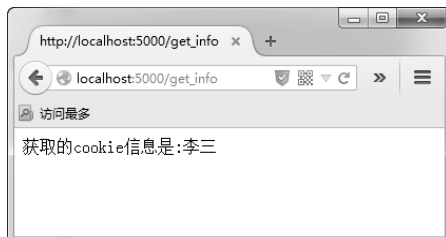


图 17.15 获取 cookie 信息

使用 session 会话同样可以完成以上功能。

【实例 17-7】演示了一个使用 session 跟踪用户的实例，代码如下：

```
# -*- encoding:utf-8 -*-
import flask

html_txt = """
<!DOCTYPE html>
<html>
    <body>
        <h2>收到 GET 请求</h2>
        <a href='/get_info'>获取会话信息</a>
    </body>
</html>
"""

app = flask.Flask(__name__)

@app.route('/set_info/<name>')
def set_cks(name):
    name = name if name else 'anonymous'
    flask.session['name'] = name
    return html_txt

@app.route('/get_info')
def get_cks():
    name = 'name' in flask.session and flask.session['name']
    if name:
        return '获取的会话信息是:' + name
    else:
        return '没有相应会话信息。'

if __name__ == '__main__':
    app.secret_key = 'dfadff#$#5dgfdgssgfgsfgr4$T%^'
    app.run(debug=True)
```

#设置 session

#获取 session

#设置 cookie 密钥

【代码说明】代码基本框架与实例 17-6 相同，只是保存状态信息的方法是 session。

【运行效果】设置会话信息的页面如图 17.16 所示，而获取会话信息的页面如图 17.17 所示。

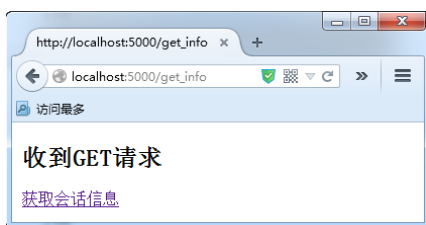


图 17.16 设置会话信息

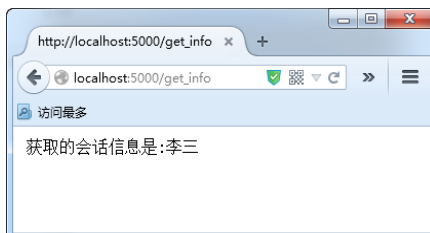


图 17.17 获取会话信息



17.2.6 使用静态文件资源与页面文件

网站中的网页必然要用到图片、CSS、js 等文件，而 Flask 框架都已经为你准备好了。要使用这种静态文件，只要在模板文件中使用以下语句：

```
url_for('static', filename='my.jpg')
```

它会生成一个网址（/static/my.jpg），要求静态文件保存在当前目录的 static 文件夹下。

另一方面，前文介绍的服务器的页面内容都是以字符串的形式保存在服务器程序中的。这种使用方法肯定是不好的，既不利于维护，也不利于程序员分工。其实，你可以使用 flask.render_template() 来调用当前目录下的 templates 子目录中的 HTML 文件，其具体形式为：

```
flask.render_template('name.html', name='name')
```

其中参数如下：

- name.html 是要返回页面内容的文件名；
- name 传递变量 name 的值为字符串'name'，供其在页面文件内容中输出相关信息。

【实例 17-8】演示了使用独立 HTML 文件的 URL 服务，并在其中引用了服务器中的图片文件，代码如下：

```
# -*- encoding:utf-8 -*-
import flask

app = flask.Flask(__name__)

@app.route('/hello')
def hello():
    return flask.render_template('index.html')          #渲染 index.html 文件

if __name__ == '__main__':
    app.run(debug=True)
```

你还应该准备相应的静态文件到当前目录的对应子目录中：

```
static/test.jpg
templates/index.html
```

其中 templates/index.html 文件的内容如下：

```
<!DOCTYPE html>
<html>
  <body>
    <img src = "{{ url_for('static',filename='test.jpg') }}" />
  </body>
</html>
```



注意 此处使用了 jinja2 的模板渲染库，它已经在安装 Flask 时作为依赖库被安装了。如果想了解具体的使用方法，你可以参考相关资料。

【代码说明】程序代码还是很简单的，只不过在业务函数中返回了 flask.render_template() 方法的结果来渲染 index.html 页面文件。index.html 文件中使用了 url_for('static',filename='test.jpg') 来生成静态资源图片文件 test.jpg 的链接。

【运行效果】如图 17.18 所示，经过渲染的页面中显示了准备的图片。

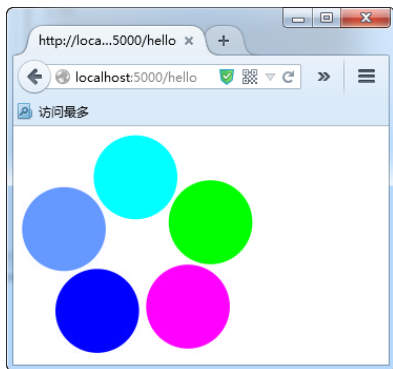


图 17.18 渲染模板文件，访问静态资源

17.2.7 接收上传文件

使用 Flask 框架编写上传文件的服务器端也很简单，它与处理 GET 或 POST 参数具有相似的地方。客户端上传的文件相关信息会被保存在 `flask.request.files` 对象中，通过这个对象，可以获取上传的文件名和文件对象，然后调用文件对象的 `save()` 将文件保存到指定的目录中即可。

【实例 17-9】 演示了一个文件上传的基本例子，代码如下：

```
# -*- encoding:utf-8 -*-
import flask

app = flask.Flask(__name__)

@app.route('/upload',methods=['GET','POST'])          #请求的 GET、POST 映射同一函数
def upload():
    if flask.request.method == 'GET':
        return flask.render_template('upload.html')  #请求方法为 POST 时返回上传页面
    else:
        file = flask.request.files['file']           #获取文件对象
        if file:                                     #如果文件不是空
            file.save(file.filename)                  #保存文件（以传来的文件名）
            return '上传成功！'

if __name__ == '__main__':
    app.run(debug=True)
```

而上传文件页面的模板文件如下：

```
<!DOCTYPE html>
<html>
  <body>
    <h2>请你选择一个文件上传</h2>
    <form method='post' enctype='multipart/form-data'>
      <input type='file' name='file' />
      <input type='submit' value='上传' />
    </form>
  </body>
</html>
```

【代码说明】 代码中只定义了一个文件上传的业务函数 `upload()`，同时接受 GET 请求和 POST 请求。GET 请求时返回上传页面，POST 请求时获取上传文件并保存在当前目录下。

【运行效果】 上传文件的页面如图 17.19 所示，页面中显示了已经选择一个图片文件上传，上传成功的页面如图 17.20 所示。

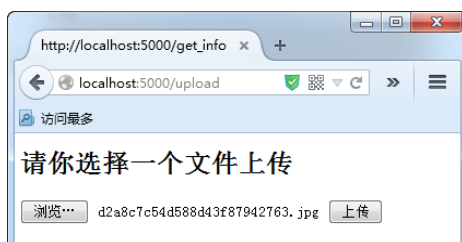


图 17.19 文件上传页面

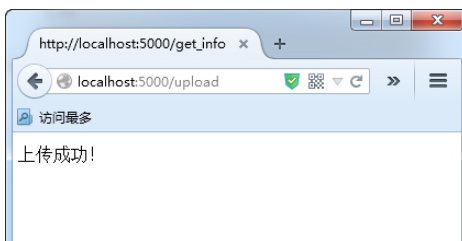


图 17.20 文件上传成功页面



本上传文件实例在应用时应添加一些校验文件类型及文件名称的功能。

17.2.8 在 Flask 框架中使用数据库

上一章介绍了数据库的相关操作，在网站开发中就是要大量使用数据库系统的；数据库系统是网站系统的主要构成部分之一，它主要用来保存网站的相关信息和用户的有关信息。所以在网站开始中要经常业务方法中访问数据库。而访问数据库一般都建立连接、查询数据、关闭连接。

在 Flask 框架中提供了一种只需要定义一次连接和定义一次释放连接，之后就可以在所有业务函数中直接使用数据连接来查询数据库。这就需要使用 Flask 框架中的两个装饰器和一个 g 对象。

两个装饰器分别是 `before_request()` 和 `teardown_request()`，被 `before_request()` 装饰的函数会在每个请求之前调用，而被 `teardown_request()` 装饰的函数会在每个请求结束之后调用。此外还有一个 `after_request()` 装饰器，只不过它在业务函数中引发错误时，它装饰的函数不会被执行。

Flask 提供了特殊的 g 对象，这个对象与每个请求是一一对应的，并且只在函数内部有效。不要在其他对象中存储类似信息，因为在多线程环境下无效。这个特殊的 g 对象会在后台神奇地工作，保证系统正常运行。

将刚才介绍的装饰器和 g 对象结合起来，就可以实现在所有的请求处理方法中直接使用数据库，而不用在请求处理方法中去重复地编写数据库连接与断开的代码。其基本使用代码如下：

```
@app.before_request                                #应用装饰器，每个请求开始时运行被装饰函数
def before_request():
    g.db = connect(DBNAME)                          #连接 sqlite3 数据库

@app.teardown_request                                #应用装饰器，每个请求退出时运行被装饰函数
def teardown_request(e):
    db = getattr(g, 'db', None)
    if db:
        db.close()                                  #关闭数据库连接
    g.db.close()
```



注意 使用这种装饰器只要在整个项目中定义一次,可在整个网站程序中直接应用,而不用多次定义。

这样,你可以在整个网站程序的请求处理方法中使用以下形式直接使用数据库连接了:

```
cur = g.db.cursor()           #获取数据库的游标
cur.execute("sql 语句字符串") #查询数据库
cur.connection.commit()       #提交事务
cur.close()                   #关闭游标
```

【实例 17-10】演示了一个使用数据库的保存用户注册信息和验证登录的简单实例,各请求方法的主要代码如下,详细代码请参阅本书所附源代码 C17 目录中的 a17_10.py:

```
@app.route('/')           #本网站的主页 (index)
def index():
    if 'username' in session:      #会话中有用户名信息,用户已登录,显示用户信息、注销
        return "你好, " + session['username'] + '<p><a href="/logout">注销</a></p>'
    else:                          #会话中没有用户名,用户未登录,显示登录和注册
        return '<a href="/login">登录</a>,<a href="/signup">注册</a>'

@app.route('/signup',methods=['GET','POST']) #用户注册 URL
def signup():                      #用户注册的业务函数
    if request.method == 'GET':     #GET 请求,返回注册页面
        return render_template('signup.html')
    else:                           #POST 请求,进行注册操作
        name = 'name' in request.form and request.form['name']
        passwd = 'passwd' in request.form and request.form['passwd']
        if name and passwd:         #校验密码和用户名不能为空
            cur = g.db.cursor()      #获取数据库的游标
            cur.execute('insert into user (name,passwd) values (?,?)',
                          (name,passwd)) #向数据库中插入用户注册信息
            cur.connection.commit()   #提交保存插入数据
            cur.close()              #关闭游标
            session['username'] = name #设置会话
            return redirect(url_for('index')) #转向主页
        else:
            return redirect(url_for('signup')) #注册信息不全,返回注册页面

@app.route('/login',methods=['GET','POST']) #用户登录的 URL
def login():                            #用户登录的业务函数
    if request.method == 'GET':         #GET 请求
        return render_template('login.html') #返回登录页面
    else:                               #POST 请求
        name = 'name' in request.form and request.form['name']
        passwd = 'passwd' in request.form and request.form['passwd']
        if name and passwd:            #校验用户信息不能为空
            cur = g.db.cursor()         #以下查询数据库,验证用户名与密码
            cur.execute('select * from user where name=?',(name,))
            res = cur.fetchone()
            if res and res[1] == passwd:
                session['username'] = name #成功验证,设置会话
                return redirect(url_for('index')) #返回主页
            else:
                return '登录失败!'
    else:
        return '参数不全!'
```



```
@app.route('/logout')                                #用户注销的 URL
def logout():                                         #用户注销的业务函数
    session.pop('username',None)
    return redirect(url_for('index'))
```

根据网站需求，还定义了两个模板文件，它们分别为 login.html 和 signup.html，其基本代码如下：

```
#templates/login.html
<!DOCTYPE html>
<html>
    <body>
        <form method='post'>
            <input type='text' name='name' placeholder='用户名' />
            <input type='password' name='passwd' placeholder='密码' />
            <input type='submit' value='登录' />
        </form>
    </body>
</html>
#templates/signup.html 文件
<!DOCTYPE html>
<html>
    <body>
        <form method='post'>
            <input type='text' name='name' placeholder='用户名' />
            <input type='password' name='passwd' placeholder='密码' />
            <input type='submit' value='注册' />
        </form>
    </body>
</html>
```

【代码说明】本实例网站服务器代码总共不超过 80 行，但是完成了一个网站项目所需要的基本功能，即网站主页、注册页面与注册功能、登录页面与登录功能、注销功能。共定义了三个数据库相关操作的函数和四个网站服务器业务函数（index()、signup()、login()、logout()）。

【运行效果】用户访问主页而没有登录的页面如图 17.21 所示，当用户单击“注册”链接时则进入注册页面（如图 17.22 所示），用户在注册页面填写用户和密码并单击“注册”按钮时，网站就会校验并记录用户名和密码，然后将其保存到数据库中为用户登录提供凭据，成功注册则设置会话，并跳转到网站主页，显示用户的相关信息及注销链接。如果用户已经注册过本网站，则可以在主页面单击“登录”后进入登录页面（如图 17.23 所示）填写用户名和密码，服务器收到用户登录信息后并进行登录验证，成功则跳转到主页面，显示用户的相关信息及注销链接（如图 17.24 所示）。

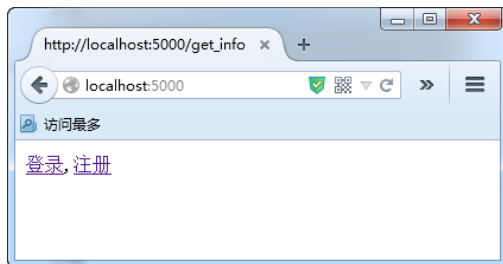


图 17.21 未登录主页面

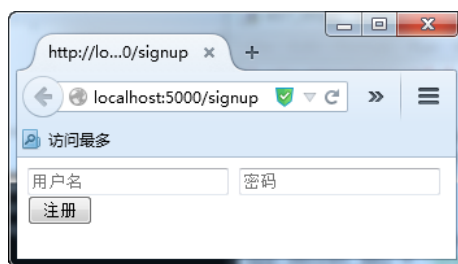


图 17.22 注册页面

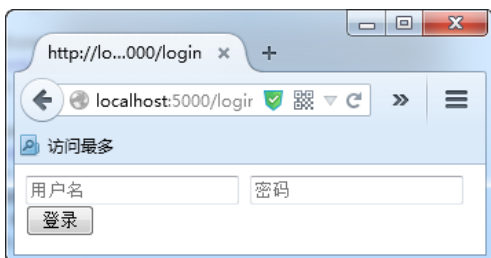


图 17.23 登录页面

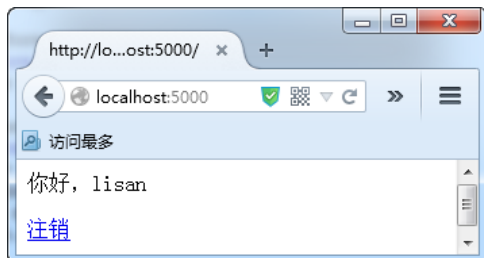


图 17.24 登录后主页面

注意

以上网站项目代码在实际应用中会将业务函数中的一些功能进行分割, 把不同的功能划分到不同的模块中, 便于程序的维护和项目的分工。

17.3 Tornado Web 框架及其应用



Tornado 也是一种比较流行的、强大的、可扩展的 Python 的 Web 非阻塞式开源服务器框架, 也是一个异步的网络库。让你能够快速简单地编写高速的 Web 应用。本节主要介绍 Tornado 框架的应用。

17.3.1 Tornado 框架简介

Tornado 是基于 Bret Taylor 和其他人员为 FriendFeed 所开发的网络服务框架, 当 FriendFeed 被 Facebook 收购后得以开源。Tornado 在设计之初就考虑到了性能因素, 旨在解决 C10K 问题, 这样的设计使得其成为一个拥有非常高性能的框架。此外, 它还拥有处理安全性、用户验证、社交网络以及与外部服务 (如数据库和网站 API) 进行异步交互的工具。

自 2009 年发布以来, Tornado 已经获得了很多社区的支持, 并且在一系列不同的场合得到应用。除 FriendFeed 和 Facebook 外, 还有很多公司在生产上转向 Tornado, 包括 Quora、Turntable.fm、Bit.ly、Hipmunk 以及 MyYearbook 等。

其主要特性有:

- 非阻塞式服务器;
- 速度相当快;
- 并发打开数千连接;
- 支持 WebSocket 连接。

Tornado 库可以大体上分为四个部分:

- tornado.Web: 创建 Web 应用程序的 Web 框架;
- HTTPServer 和 AsyncHTTPClient: HTTP 服务器与异步客户端;
- IOLoop 和 IOStream: 异步网络功能库;
- tornado.gen: 协程库。

从编程风格上来看, 使用 Tornado 框架编写 Web 服务器端更像是面对对象编程。

17.3.2 Tornado 框架初识

要使用 Tornado 这个 Web 框架, 也必须先安装它。其安装方法与本书前面所述的各种第三方库的安装方法相同:

```
pip install tornado
```



到 <https://pypi.python.org/packages/source/t/tornado/tornado-4.1.tar.gz> 网址下载 tornado 的源码，解压缩后，在命令提示符下对该子目录执行以下命令：

```
python setup.py install
```

用 Tornado 框架来编写 Web 服务器端程序就是通过继承 `tornado.Web.RequestHandler` 类，并编写 `get()`、`post()` 业务方法，以实现对客户指定 URL 的 GET 请求和 POST 请求的回应。然后启动框架中提供的服务器以等待客户端连接、处理相关数据并返回请求信息。

用 Tornado 框架编写 Web 服务器的基本代码框架如下：

```
import tornado.ioloop                                #导入相关模块
import tornado.Web

class MainHdl(tornado.Web.RequestHandler):            #定义类（继承 RequestHandler）

    def get(self):                                     #定义 GET 请求业务方法
        pass

    def post(self):                                    #定义 POST 请求业务方法
        pass

app = tornado.Web.Application([                        #调用 tornado 初始化应用
    (r'/',MainHdl),
    .....
if __name__ == '__main__':
    app.listen(8888)                                  #服务器监听服务端 8888
    tornado.ioloop.IOLoop.instance().start()          #启动服务器，等待客户端连接
```

通过以上框架可以看出，用 Tornado 框架编写服务器端程序的代码结构是非常清晰的。其基本工作就是编写相关的业务处理类，并将它们和某一特定的 URL 映射起来，Tornado 框架服务器收到对应的请求后进行调用。

一般来说简单的网站项目可以把所有的代码放入同一个模块之中，但为了维护方便，可按照功能将其划分到不同的模块中，其一般模块结构（目录结构）如下：

```
proj\
    manage.py          #服务器启动入口
    settings.py        #服务器配置文件
    url.py             #服务器 URL 配置文件
    handler\
        login.py       #相关 URL 业务请求处理类
    db\                #数据库操作模块目录
    static\            #静态文件存放目录
        js\            #JS 文件存放目录
        css            #CSS 样式表文件目录
        img\          #图片资源文件目录
    templates\         #网页模板文件目录
```

这种目录结构也正符合了目前流行的网站设计风格——MVC 模式，即模板、视图、控制器模式。

【实例 17-11】 演示了一个使用 Tornado 框架编写的最基本的服务器程序，代码如下：

```
# -*- encoding:utf-8 -*-
import tornado.ioloop
import tornado.Web

class MainHdl(tornado.Web.RequestHandler):            #自定义类
```

```

def get(self):
    self.write('你好, 我是 Tornado!')

app = tornado.Web.Application([
    (r'/', MainHdl),
    ], debug=True)

if __name__ == '__main__':
    app.listen(8888)
    tornado.ioloop.IOLoop.instance().start()

```

回应 GET 请求方法
URL 映射列表 (可有多条)
服务器监听 8888 端口
启动服务器

【代码说明】由以上代码可以看出, 用 Tornado 框架写一个基本的 Web 服务器端程序也不过十行代码且代码结构清晰。首先要导入 Tornado 相关模块, 然后自定义 URL 的响应业务方法 (GET、POST 等), 其次是实例化 Tornado 模块中提供的 Application 类, 并传送 URL 映射列表及有关参数, 最后启动服务器即可。在命令提示符下的对应子目录中执行:

```
python a17_11.py
```

如果没有语法错误的话, 服务器就已经启动并等待客户端连接了。

【运行效果】服务器运行后, 在浏览器地址栏中输入 `http://localhost:8888` 就可以访问服务器, 看到默认主页页面了。本例页面如图 17.25 所示。

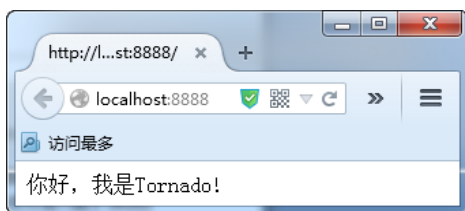


图 17.25 基本 Tornado 服务器页面

17.3.3 请求参数的获取

如 17.2 节所述, 客户端请求的参数有 URL 中的参数、GET 请求中的参数和 POST 请求中的参数。

在 Tornado 框架中, 要获取 URL 中包含的参数, 与 Flask 框架有相似之处, 它们都是在 URL 定义中定义获取参数, 并在对应的业务方法中给出相应的参数名获取。

Tornado 框架 URL 定义字符串中, 使用正则表达式来匹配 URL 及 URL 中的参数, 比如:

```
(r"uid/([0-9]+)", UserHdl)
```

这种形式的 URL 字符串定义可以接受形如 “uid/” 后跟一位或多位数字的客户端 URL 请求。

对应以上 URL 定义, 可用以下方式定义 `get()` 方法:

```

def get(self, uid):
    pass

```

这样, 当匹配的 URL 请求到来时, 会截取属于正则组匹配的部分, 传递给 `get()` 方法, 从而把数据传递给 `uid` 变量, 在 `get()` 方法中得到使用。

【实例 17-12】演示了在 GET 方法中获取 URL 中参数的基本实例, 代码如下:

```

# -*- encoding:utf-8 -*-
import tornado.ioloop
import tornado.Web

```




```
class MainHdl(tornado.Web.RequestHandler):
    def get(self,uid='0'):
        self.write('你好,你的UID号是: %s!' % uid)

app = tornado.Web.Application([
    (r'/([0-9]+)',MainHdl),
],debug=True)

if __name__ == '__main__':
    app.listen(8888)
    tornado.ioloop.IOLoop.instance().start()
```

【代码说明】代码中使用了上述正则表达式的 URL 字符串定义及带有 uid 参数的 get() 方法。

【运行效果】服务器运行后，浏览器访问结果如图 17.26 所示。

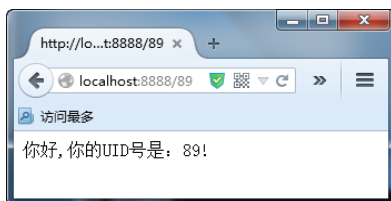


图 17.26 捕获处理 URL 中的参数

如果要获取 GET 或 POST 请求参数，则更加简单。只需要调用从 RequestHandler 类中继承来的 get_argument() 方法，其原型如下：

```
get_argument('name',default='',strip=False)
```

其中参数意义是：

- name 请求中的参数名称；
- default 指定没有获取参数时给定一个默认值；
- strip 指定是否对获取的参数进行两头去空格处理。

【实例 17-13】演示一个获取 POST 请求中参数的实例，代码如下：

```
# -*- encoding:utf-8 -*-
import tornado.ioloop
import tornado.Web

html_txt = """
<!DOCTYPE html>
<html>
<body>
    <h2>收到GET 请求</h2>
    <form method='post'>
        <input type='text' name='name' placeholder='请输入你的姓名' />
        <input type='submit' value='发送 POST 请求' />
    </form>
</body>
</html>
"""

class MainHdl(tornado.Web.RequestHandler):
    def get(self):
        self.write(html_txt)

    def post(self):
        #获取 POST 请求的参数 name
```

```

name = self.get_argument('name',default='匿名',strip=True)
self.write("你的姓名是:%s" % name)

app = tornado.Web.Application([
    (r'/get',MainHdl),
],debug=True)

if __name__ == '__main__':
    app.listen(8888)
    tornado.ioloop.IOLoop.instance().start()

```

【代码说明】在上述代码中，服务器收到 GET 请求时，返回一个带有表单的页面内容；当用户填写自己的姓名，并单击“发送 POST 请求”时，将用户输入的姓名以 POST 参数形式发送到服务器端。最后服务器端调用 `get_argument()` 方法来获取和处理它。

【运行效果】发送“localhost:8888/get”的 GET 请求时，返回页面如图 17.27 所示，而当用户提交时显示的页面如图 17.28 所示。



图 17.27 “/get” URL 的 GET 请求页面



图 17.28 “/get” URL 的 POST 请求页

17.3.4 用 cookie 与安全 cookie 跟踪客户

Tornado 框架提供了直接操纵 cookie 和安全 cookie 的方法。安全的 cookie 就是存储在客户端的 cookie，是经过加密的，客户端只能查看到加密后的数据。使用 cookie 和安全 cookie 的基本原型方法如下：

<code>set_cookie('name',value)</code>	#设置 cookie
<code>get_cookie('name')</code>	#获取 cookie 值
<code>set_secure_cookie('name',value)</code>	#设置安全 cookie 值
<code>get_secure_cookie('name')</code>	#获取安全 cookie 值
<code>clear_cookie('name')</code>	#清除名为 name 的 cookie 值
<code>clear_all_cookies()</code>	#清除所有 cookie



注意

要使用安全 cookie，必须为 Application 类提供 `cookie_secret` 参数，以给出加密的密钥。

【实例 17-14】演示了一个在不同页面设置与获取 cookie 值的实例，代码如下：



```
# -*- encoding:utf-8 -*-
import tornado.ioloop
import tornado.Web

class ScookHdl(tornado.Web.RequestHandler):
    def get(self):
        odn_cookie = tornado.escape.url_escape("未加密 COOKIE 串") #URL 编码
        self.set_cookie('odn_cookie',odn_cookie) #设置一般 cookie
        self.set_secure_cookie('scr_cookie',"SCURE_COOKIE") #设置安全 cookie
        self.write("<a href='/gcook'>查看设置的 COOKIE</a>")

class GcookHdl(tornado.Web.RequestHandler):
    def get(self):
        odn_cookie = self.get_cookie('odn_cookie') #获取一般 cookie 值
        odn_cookie = tornado.escape.url_unescape(odn_cookie) #反 URL 编码
        #获取安全 cookie 值
        scr_cookie = self.get_secure_cookie('scr_cookie').decode('utf-8')
        self.write("普通 COOKIE:%s,安全 COOKIE:%s" % (odn_cookie,scr_cookie))

app = tornado.Web.Application([
    (r'/scook',ScookHdl),
    (r'/gcook',GcookHdl),
],cookie_secret='abcdkdkk###$34323sdDsdffsf#23')

if __name__ == '__main__':
    app.listen(8888)
    tornado.ioloop.IOLoop.instance().start()
```

【代码说明】代码中共定义了两个类，分别用来设置 cookie 和获取 cookie，就是应用前面介绍的方法。

【运行效果】当用户访问“/scook”时会设置 cookie，其页面如图 17.29 所示；当用户单击页面中的“查看设置的 COOKIE”链接时，会访问“/gcook”，从而显示出 cookie 中设置的值，如图 17.30 所示。

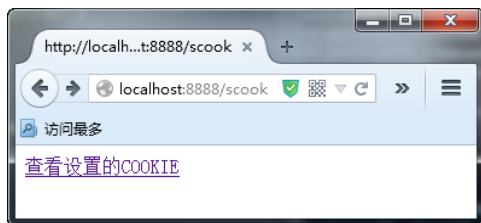


图 17.29 设置 cookie 页面

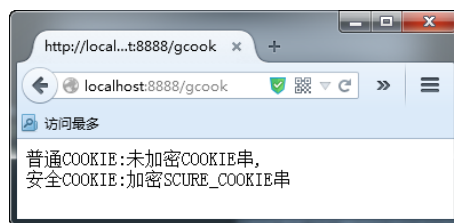


图 17.30 显示设置的 cookie 页面



注意

因字符串编码的问题，在设置 cookie 字符串中有中文字符时，要使用 tornado.escape 模块中的 URL 编码与解码，否则会出现乱码的现象。

此外，Tornado 框架中并不提供 session 功能，你要使用就必须自己实现功能。

17.3.5 URL 转向与静态文件资源

在 Tornado 框架的 Web 编程中，也可以实现与 Flask 中相同的 URL 转向的功能。Tornado 框架中有两种方法可以实现 URL 转向：

- redirect(url) 在业务逻辑中转向 URL；
- RedirectHandler 实现某个 URL 的直接转向。

RedirectHandler 类的具体使用形式为:

```
(r'/aaa',tornado.Web.RedirectHandler,dict(url='/abc'))
```

【实例 17-15】 演示了两种 URL 转向的实例,代码如下:

```
# -*- encoding:utf-8 -*-
import tornado.ioloop
import tornado.Web

class DistHdl(tornado.Web.RequestHandler):
    def get(self):
        self.write("被转向的目的页面!")

class SrcHdl(tornado.Web.RequestHandler):
    def get(self):
        self.redirect('/dist') #在业务逻辑中转向

app = tornado.Web.Application([
    (r'/dist',DistHdl),
    (r'/src',SrcHdl),
    (r'/rdrt',tornado.Web.RedirectHandler,{ 'url':'/src'}) #直接转向
])

if __name__ == '__main__':
    app.listen(8888)
    tornado.ioloop.IOLoop.instance().start()
```

【代码说明】 代码中定义了两个类, DistHdl 作为转向的目标 URL 请求处理器, SrcHdl 是转向处理器, 当访问指向这个业务类时, 会被转向到‘dist’网址。最后, 在 Application 类中定义一个直接转向, 只要访问‘rdrt’就会直接转向到‘src’。所以, 有趣的是, 如果你试图访问‘rdrt’URL, 会转向‘src’, 最终转向‘dist’。

【运行效果】 运行的结果无论是访问‘rdrt’, 还是访问‘src’, 最后都会如图 17.31 所示。

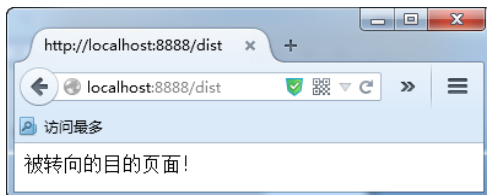


图 17.31 转向的目的页面

Tornado 框架也支持在网站的页面中直接使用静态的资源文件, 如图片、JS 脚本、CSS 样式表等。如果要使用静态文件资源, 需在 Application 类初始化时提供“static_path”参数。

【实例 17-16】 演示了在网站页面中引用图片的实例, 代码如下:

```
# -*- encoding:utf-8 -*-
import tornado.ioloop
import tornado.Web

class SttHdl(tornado.Web.RequestHandler):
    def get(self):
        self.write("<img src='/static/torn.jpg' />") #使用了本网站图片

app = tornado.Web.Application([
    (r'/stt',SttHdl),
    ],static_path='./static') #提供了 static_path 参数
```



```
if __name__ == '__main__':
    app.listen(8888)
    tornado.ioloop.IOLoop.instance().start()
```

【代码说明】代码中‘/stt’请求返回的 HTML 代码中是一个 img 标签，引用本网站中的一幅图片。在初始化 Application 类时提供了 static_path 参数，以指明静态资源的目录。

注意 此处的静态资源目录是相对于运行时所在的路径目录，所以运行服务器命令行时应进入对应的目录，否则无法找到静态资源。你可以使用绝对路径来排除此问题。

【运行效果】如图 17.32 所示，页面中显示了一幅图片。

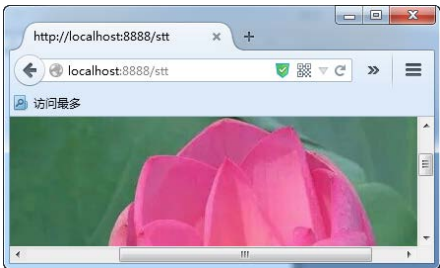


图 17.32 使用图片静态资源

17.3.6 Tornado Web 框架应用举例



从上面对 Tornado 框架介绍中可以看出，使用它开发一个网站项目还是比较容易和快速的。本节以开发一个基本的互动学习系统为例，演示和讲解用 Tornado 框架开发网站的基本过程。

这个智动交互式学习系统只有两个主要页面，其中主页面用来登录、学习以及管理员管理（如图 17.33 所示），另一个页面用于用户注册。它实现的主要功能是教师先在主页面添加学习主题，即某节课学习的相关问题并提交（问题直接为文本形式），如图 17.34 所示。当学生未登录时可以显示教师发布的学习主题以及每个主题的学习问题，如果学生登录，则可以回答问题。同时，还可以在主页中进行互动问题的讨论，如图 17.35 所示。

智 动 交 互 式 学 习 系 统

你还没有登录？姓名

学习主题

1. 2. 2I/O接口的基本概念（2014-10-24）

主板上的主要部件(1.1.3)-2014-9-26
1.1.4外存储器2014-10-12
1.2.1常见输入输出接口（2014-10-21）
1.2.2I/O接口的基本概念（2014-10-24）
1.2.3-1输入输出信息传送控制方式（2014-10-29）
1.2.3程序中断方式与DMA方式(2014-11-04)
1.2.3通道方式(2014-11-09)
1.2.4外围设备1(2014-12-02)
1.2.4外围设备2-显示器(2014-12-02)
1.2.4外围设备3-打印机(2014-12-16)
2.1.1数制数制间转换(2014-12-23)
2.1.1数制数制间转换2(2014-12-28)
2.1.1数制与数制间的转换3(2015-01-06)
2.1.2数值信息在计算机中的表示1

一、填空题

1、接口即（ ），具体是指（ ）和（ ）、（ ）之间通过（ ）进行连接的（ ）。

2、接口部件在它动态连接的两个部件之间起着（ ）的作用。

3、不同的I/O设备都有各自（ ）、（ ）和（ ）。

4、主机与外部设备数据格式上也不相同：主机采用（ ）表示数据，而外部设备一般采用（ ）。

5、CPU通过对（ ）内容的读取和检测可以确定I/O接口当前的工作状态（也是外设的工作状态）。

图 17.33 未登录时的首页

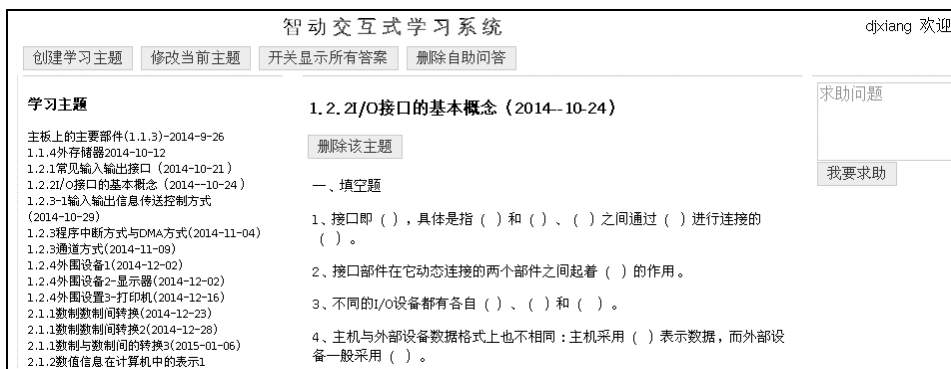


图 17.34 老师登录后的首页

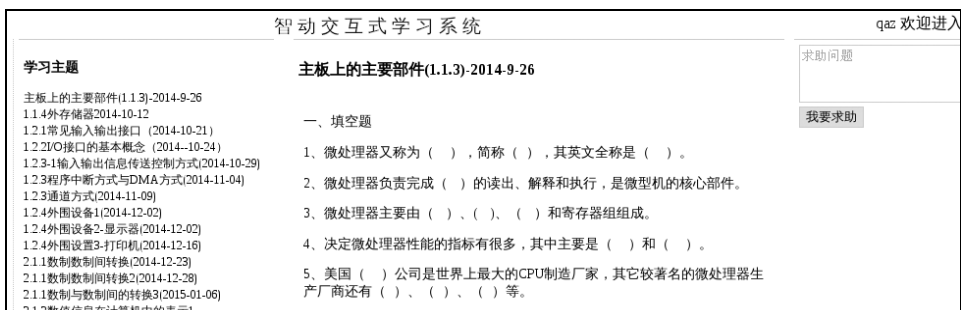


图 17.35 学生登录后的首页

该系统使用的数据库为 Python 内嵌的 SQLite3 数据库，如前所述：它是标准库自带的，无须安装即可使用。另外，还使用了一个速度比较快的第三方模板库 mako 来渲染网页。本来 Tornado 框架是自带一个基本的模板渲染系统的。而这里通过轻松地定制，就把其原有的功能进行了替换。

本学习系统需要存储的数据并不是很多，主要有用户注册信息、学习主题与问题、学生对问题的回答、学生们互动提问和学生或教师对互动提问的回答。因此，在数据库中主要建立的表及字段属性如表 17.1 至表 17.5 所示。

表 17.1 stu_sbjct (学习主题表)

字段名	类型	说明
id	integer	序号(主键)
title	varchar(500)	标题
qstn	text	问题内容
openothr	integer	是否显示其他人答案，默认 0 表示不显示

表 17.2 stu_answrs (学生答案)

字段名	类型	说明
id	integer	序号(主键)
sbjct_id	integer	答案的主题号(外键)
stu_id	integer	答案的学生号(外键)
answer	text	答案的内容
answer_time	timestamp	回答的时间



表 17.3 ask_hlps（互动提问的问题表）

字段名	类 型	说 明
id	integer	序号（主键）
stu_id	integer	提问的学生号（外键）
qstn	text	提问的内容
ask_time	timestamp	提问的时间

表 17.4 hlp_answrs（互动提问讨论）

字段名	类 型	说 明
id	integer	序号（主键）
ask_id	integer	问题的主题号（外键）
hlper_id	integer	答案的用户号（外键）
answr	text	答案的内容
answr_time	timestamp	回答的时间

表 17.5 stds（注册用户表）

字段名	类 型	说 明
id	integer	序号（主键）
name	varchar(8)	姓名
psswd	varchar(256)	密码
usertype	integer	用户类型
ipaddr	varchar(20)	用户登录的 IP 地址

数据库操作的相关功能模块被放在同一个文件（模块）中，详细代码请看本书所附源代码 C17 目录中的 stir-mentality 子目录下的 db_mgr.py 源代码文件。

首先自定义一个带上下文管理功能的数据库操作类（实现了上下文管理器的协议方法 `__enter__()` 和 `__exit__()`），以方便在实际数据库操作中使用，其代码如下：

```
class MySQLiteDb(object):                                #定义数据库操作类
    """Sqlite3 Db Class"""
    def __init__(self, dbname="mys.db"):                  #构造方法（传入数据库名）
        self.dbname = dbname
        self.con = None
        self.curs = None

    def getCursor(self):                                  #定义获取数据连接的游标
        self.con = sqlite3.connect(self.dbname)
        if self.con:
            self.curs = self.con.cursor()

    def closeDb(self):                                    #定义关闭数据库连接
        if self.curs:
            self.curs.close()
        if self.con:
            self.con.commit()
            self.con.close()

    def __enter__(self):                                  #上下文管理器协议方法
        self.getCursor()
        return self.curs
```

```

#上下文管理器协议方法
def __exit__(self, exc_type, exc_val, exc_tb):
    if exc_val:
        print("Exception has generate: ",exc_val)
        print("Sqlite3 execute error!")
    self.closeDb()

```

其次定义一个数据库初始化的函数，供网站程序启动时调用，其主要功能就是创建前文所述的数据库表，代码如下：

```

def initDb(db):
    crtSql = (
        '''
        create table stu_sbjet
        (id integer primary key autoincrement not null,
        title varchar(500) not null,
        qstn text,
        openothr integer default 0)
        ''',
        '''
        create table stu_answrs(
        id integer primary key autoincrement not null,
        sbjct_id integer,
        stu_id integer,
        answr text,
        answr_time timestamp default current_timestamp
        )
        ''',
        '''
        create table stds
        (
        id integer primary key autoincrement not null,
        name varchar(8),
        psswd varchar(256),
        usertype integer,
        ipaddr varchar(20)
        )
        ''',
        '''
        create table ask_hlps
        (
        id integer primary key autoincrement not null,
        stu_id integer,
        qstn text,
        ask_time timestamp default current_timestamp
        )
        ''',
        '''
        create table hlp_answrs
        (
        id integer primary key autoincrement not null,
        ask_id integer,
        hlper_id integer,
        answr text,
        answr_time timestamp default current_timestamp
        )
        ''')
    for sql in crtSql:
        db.execute(sql)

```

#定义初始化数据库方法（创建表）
#创建所有表的 SQL 语句
#循环执行 SQL 语句以创建表

最后，为每个数据库表定义了一个操作数据库的类，即数据业务方法。此处列出学生的用



户表 (stds) 的业务方法类 (其他数据库表的操作类请参阅本书所附源文件的 C17 目录中 stir-mentality 子目录下的 db_mgr.py 源代码文件), 其中定义了初始化类的构造方法、保存用户名等数据到数据库的 save() 方法、验证用户名及密码检测登录的 isRgstr() 方法、查询已注册过的用户等方法, 这样在定义响应用户的 URL 请求的方法中, 就可以直接实例化此类并调用其数据库业务方法, 而不用将代码全部写入 URL 请求方法中, 以达到分离代码的目的, 方便代码的维护。其代码如下:

```
class Stu(object):                                #定义用户数据操作类
    """class for stds"""
    def __init__(self, id=0, name='', psswd='', usertype=0, ipaddr=''):
        self.id = id
        self.name = name
        self.psswd = psswd
        self.usertype = usertype
        self.ipaddr = ipaddr

    def save(self):                                #持久化类数据方法
        if self.name and self.psswd:
            with MySQLiteDb() as db:
                db.execute(
                    "insert into stds (name,psswd,usertype,ipaddr) values (?,?,,?)",
                    (self.name,self.psswd,self.usertype,self.ipaddr)
                )
            return True

    def isRgstr(self):                              #验证登录的方法
        with MySQLiteDb() as db:
            res = db.execute(
                "select * from stds where name=? and psswd=?",
                (self.name,self.psswd)
            )
            res = res.fetchall()
            res_ip = db.execute(
                "select * from stds where ipaddr=? and name != ?",
                (self.ipaddr,self.name)
            )
            res_ip = res_ip.fetchall()
            if res and not res_ip:
                with MySQLiteDb() as db:
                    db.execute("update stds set ipaddr=? where name=? and psswd=?",
                                (self.ipaddr,self.name,self.psswd)
                            )
                return res[0]
            else:
                return False

    def getStuName(self, stu_id):                   #根据用户 id 获取用户名方法
        with MySQLiteDb() as db:
            res = db.execute(
                "select * from stds where id=?", (stu_id,)
            )
            res = res.fetchall()
            if res:
                return res[1]
            else:
                return ''

    def had_name(self):                              #查询数据表中指定用户名是否被注册过
        with MySQLiteDb() as db:
            res = db.execute("select * from stds where name=?", (self.name,))
            res = res.fetchall()
```

```

if res:
    return True
else:
    return False

```

有了这些数据处理类，服务器的主程序就得到大量简化，当程序需要操作或查询数据库中的数据时，直接调用这个模块中的相关函数或类实例的方法即可。

该学习系统的主程序代码请参阅本书附源文件的 C17 目录中 stir-mentality 子目录下的 manger.py 源代码文件。

主程序代码中首先自定义了继承 tornado.Web.RequestHandler 类的 BaseHandler 类，并通过重载其 render() 方法替换其模板引擎为 mako，其代码如下：

```

class BaseHandler(RequestHandler):           #定义继承 RequestHandler 类
    """ Use mako template system """         #用 mako 模板库替换 Tornado 库自带模板
    def initialize(self,lookup=LOOK_UP):
        self._lookup = lookup

    def render(self,filename,kwargs={}):      #重载 render 方法实现模板库替换
        env_kwargs = dict(
            handler=self,
            request=self.request,
            current_user=self.current_user,
            locale=self.locale,
            _=self.locale.translate,
            static_url=self.static_url,
            xsrf_form_html=self.xsrf_form_html,
            reverse_url=self.application.reverse_url)
        env_kwargs.update(kwargs)
        try:
            myTemplate = self._lookup.get_template(filename)
            self.finish(myTemplate.render(**env_kwargs))
        except:
            print('Did not find template file:',filename)
            self.send_error(404)

    def get_current_user(self):               #重载方法，实现检查用户是否登录
        return self.get_secure_cookie('name')

```

其后的大部分代码都是定义映射到指定 URL 的处理器类以响应浏览器客户端的请求，通过调用数据库业务方法，实现数据的持久化和从数据库中查询获取数据。通过把数据库的相关操作移至一个功能模块，主程序的代码得以大量简化而且结构清晰。

此外，其他的如 JavaScript、CSS 以及模板文件，请参阅随书资料。

17.4 小结

本章首先介绍了 Web 开发的基本原理，之后分别介绍了目前流行的 Python Web 框架 Flask 和 Tornado。详细介绍了两个框架的基本结构和使用方式，并在最后给出了使用 Tornado 框架编写的一个智能交互式学习系统小型网站项目。通过学习本章的内容，你应掌握 Web 工作基本原理、常见 Web 开发的基本框架和开发方式。

17.5 本章习题

一、简答题

1. 简述 Web 的工作原理。



2. Flask Web 框架有什么特点?
3. URL 参数传递和 GET/POST 参数传递有什么区别与联系?
4. Tornado Web 框架有什么特点?

二、实验题

1. 使用 Flask Web 框架实现一个 Web 版的通讯录，数据库可以自由选用，通讯的字段主要有姓名、电话、手机号、电子邮箱、QQ，可以实现添加、存储和查找。

2. 使用 Tornado Web 框架实现一个 Web 版的多用户通讯录，用户可以注册、登录，并使用通讯录，数据库可以自由选用，通讯的字段主要有姓名、电话、手机号、电子邮箱、QQ，可以实现添加、存储和查找，并且可以实现通讯录的分组。

第 18 章 数据结构基础

数据结构是用来描述一种或多种数据元素之间的特定关系，算法是程序设计中
对数据操作的描述，数据结构和算法组成了程序。对于简单的任务，只要使用编程
语言提供的基本数据类型就足够了。而对于较复杂的任务，就需要使用比基本
的数据类型构造更加复杂的数据结构了。

本章内容包括：

- 用 Python 操作表；
- 用 Python 操作栈；
- 用 Python 操作队列；
- 用 Python 操作树；
- 用 Python 操作图；
- 用 Python 进行查找；
- 用 Python 进行排序。

18.1 表、栈和队列

表、栈和队列都是基本的线性数据结构。基于 Python 设计良好的数据结构，
其列表就可以当作表来使用，而且列表的某些特性与链表相似，因此在 Python 中
表的实现非常简单。对于栈和队列，则可以自己编写程序来构建。



18.1.1 用列表来创建表

表是最基本的数据结构，在 Python 中可以使用列表来创建表。而在 C 语言中，一般使用
数组来创建表。由于使用数组创建表，对表中的元素进行插入和删除操作的开销较大。当插入
一个元素时，要先将该元素后的所有元素，从最后一个元素开始依次向后移动一个位置。完成
元素移动后，再将元素插入到数组中。同样，要删除表中的元素时，首先删除元素，然后将位
于该元素之后的元素从前向后，依次向前移动一个位置。

如果一个表含有的元素较多，而要进行插入或删除的位置又比较靠近表的前端，则移动表
中元素的操作将耗费大量的时间。为了减少插入和删除元素的线性开销，于是就出现了使用链
表代替表。在 C 语言中，链表中不仅保存数据，还保存了指向下一个元素的指针，如图 18.1
所示。当进行插入操作时，要先将位于插入元素前的元素的指针赋值给插入元素。完成赋值后
再将插入元素的地址赋值给位于其前的元素，如图 18.2 所示。当进行删除元素时，只需将要删
除元素的指针赋值给其前边的元素，如图 18.3 所示。

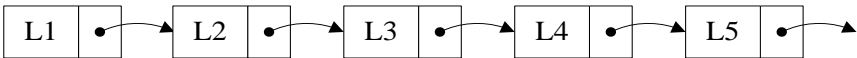


图 18.1 链表

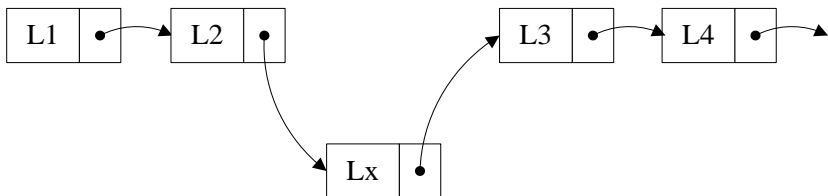


图 18.2 链表的插入操作

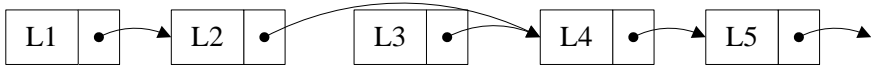


图 18.3 链表的删除操作

使用链表可以降低插入删除、元素的线性开销。然而，由于链表中不仅存储了数据，而且还保存了指向下一个元素的指针，因而使用链表将占用更大的存储空间。

而在 Python 中，列表本身就提供了插入和删除操作。因此，在 Python 中列表也可以充当链表使用，而不用自己另外编写程序来构建。

还有一种链表被称之为双向链表，如图 18.4 所示。双向链表中不仅保存了指向下一元素地址的指针，而且还保存了指向其上一个元素地址的指针。相对于单向链表，双向链表需要占用更多的存储空间，但使用双向列表可以完成正序和倒序扫描链表。



图 18.4 双向链表

18.1.2 自定义栈数据结构

栈可以看作在同一位置上进行插入和删除的表，这个位置一般被称为栈顶。栈的基本操作是进栈和出栈，栈可以看作一个容器，如图 18.5 所示。先入栈的数据保存在容器底部，后入栈的数据保存在容器顶部。在出栈的时候，后入栈的数据先出，而先入栈的数据则后出，因此栈有一个特性叫作后进先出（LIFO）。

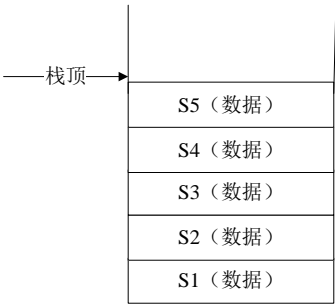


图 18.5 栈

在 Python 中，仍然可以使用列表来存储堆栈数据。通过创建一个堆栈类来实现对堆栈进行操作的方法。例如进栈 PUSH 方法、出栈 POP 方法，编写检查栈是否为满栈或者是否为空栈的方法等。

【实例 18-1】演示了在 Python 中创建了一个简单的堆栈结构，代码如下：

```

# -*- coding:utf-8 -*-
#
class PyStack:                                # 堆栈类
    def __init__(self, size = 20):
        self.stack = []                      # 堆栈列表
        self.size = size                     # 堆栈大小
        self.top = -1                        # 栈顶位置
    def setSize(self, size):                  # 设置堆栈大小
        self.size = size
    def push(self, element):                  # 元素进栈
        if self.isFull():
            raise StackException('PyStackOverflow')# 如果栈满则引发异常
        else:
            self.stack.append(element)
            self.top = self.top + 1
    def pop(self):                            # 元素出栈
        if self.isEmpty():
            raise StackException('PyStackUnderflow')# 如果栈为空则引发异常
        else:
            element = self.stack[-1]
            self.top = self.top - 1
            del self.stack[-1]
            return element
    def Top(self):                            # 获取栈顶位置
        return self.top
    def empty(self):                          # 清空栈
        self.stack = []
        self.top = -1
    def isEmpty(self):                       # 是否为空栈
        if self.top == -1:
            return True
        else:
            return False
    def isFull(self):                       # 是否为满栈
        if self.top == self.size - 1:
            return True
        else:
            return False

class StackException(Exception):             #自定义异常类
    def __init__(self,data):
        self.data=data
    def __str__(self):
        return self.data

if __name__ == '__main__':
    stack = PyStack()                        # 创建栈
    for i in range(10):
        stack.push(i)                       # 元素进栈
    print(stack.Top())                       # 输出栈顶位置
    for i in range(10):
        print(stack.pop())                  # 元素出栈
    stack.empty()                            # 清空栈
    for i in range(21):
        stack.push(i)                      # 此处将引发异常

```

【代码说明】代码中共定义了一个堆栈类 `PyStack` 和一个堆栈异常类 `StackException`，堆栈类有一个初始化参数，即堆栈大小，在堆栈类中定义了关于堆栈操作的相关方法。



【运行效果】运行 `pystack.py` 程序后，将输出如下所示的内容：

```
9
9
8
7
6
5
4
3
2
1
0
Traceback (most recent call last):
  File "D:\lx\l8\al8_1.py", line 55, in <module>
    stack.push(i)
File "D:\lx\l8\al8_1.py", line 12, in push
    raise StackException('PyStackOverflow')
__main__.StackException: PyStackOverflow
```

18.1.3 实现队列功能

队列与栈的结构类似，如图 18.5 所示。但不同的是队列的出队操作是在队首元素进行的删除操作。因而对于队列而言，先入队的元素将先出队。因此队列的特性可以被称为先进先出（FIFO）。

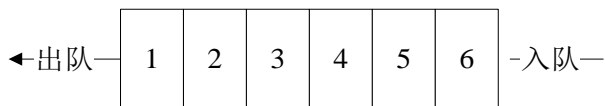


图 18.6 队列

和堆栈类似，在 Python 中同样可以使用列表来构建一个队列，并完成对队列的操作。

【实例 18-2】演示了一个创建简单的队列的实例，代码如下：

```
# -*- coding:utf-8 -*-
#
class PyQueue:                                # 创建队
    def __init__(self, size = 20):
        self.queue = []                       # 队
        self.size = size                      # 队大小
        self.end = -1                         # 队尾
    def setSize(self, size):                   # 设置队大小
        self.size = size
    def In(self, element):                     # 入队
        if self.end < self.size - 1:
            self.queue.append(element)
            self.end = self.end + 1
        else:
            raise QueueException('PyQueueFull') # 如果队满则引发异常
    def Out(self):                             # 出队
        if self.end != -1:
            element = self.queue[0]
            self.queue = self.queue[1:]
            self.end = self.end - 1
            return element
        else:
            raise QueueException('PyQueueEmpty') # 如果对为空则引发异常
    def End(self):                             # 输出队尾
```

```

        return self.end
    def empty(self):
        self.queue = []
        self.end = -1

class QueueException(Exception):
    def __init__(self,data):
        self.data=data
    def __str__(self):
        return self.data

if __name__ == '__main__':
    queue = PyQueue()
    for i in range(10):
        queue.In(i)
        print(queue.End())
    for i in range(10):
        print(queue.Out())
    for i in range(20):
        queue.In(i)
        queue.empty()
    for i in range(20):
        print(queue.Out())

```

清除队

#自定义异常类

元素入队

元素出队

元素入队

清空队

此处将引发异常

【代码说明】该实例中的代码与实例 18-1 也是相似的，定义了队列类及其相关的操作方法，以及一个异常类。

【运行效果】运行 pyqueue.py 程序后输出如下：

```

9
0
1
2
3
4
5
6
7
8
9
Traceback (most recent call last):
  File "D:\lx\18\al8_2.py", line 47, in <module>
    print(queue.Out())
  File "D:\lx\18\al8_2.py", line 23, in Out
    raise QueueException('PyQueueEmpty')
__main__.QueueException: PyQueueEmpty

```

18.2 树和图



树和前边所讲的表、栈和队列等这些线性数据结构不同，树不是线性的。在处理较多数据时，使用线性结构较慢，而使用树结构则可以提高处理速度。不过，相对于线性的表、栈和队列等线性数据结构来说，树的构建便显得较为复杂了。

18.2.1 用列表构建树

树是一种非线性的数据结构，如图 18.7 所示。之所以称之为树，是因为其形状像一颗倒置的树。每棵树都有一个根节点，在如图 18.7 所示的树中，Root 为根节点；A、B、C 为 Root 的儿子，Root 为 A、B、C 的父亲，A、B、C 为兄弟；同样，A 为 D、E 的父亲，D、E 为 A 的儿子，D、E 为兄弟；D、E 为 Root 的孙子，Root 为 D、E 的祖父。在树中，如果一个元素



没有儿子，则称之为树的叶子。

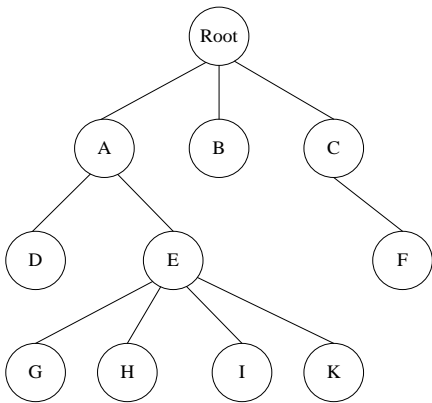


图 18.7 树

在 Python 中，树的实现可以使用列表或者类的方式。使用列表的方式较为简便，但树的构建过程较为复杂。使用类的方式构建树时，需要首先确定树中的节点所能拥有的最大儿子数。因为每个节点所拥有的儿子数量并不一定相同，因此使用类的方法将占用更大的存储空间。

【实例 18-3】演示了以列表的形式构建了如图 18.7 所示的树，代码如下：

```
# -*- coding:utf-8 -*-
#
G = [ 'G', [] ]                                # 构造叶子 G，树中每个元素都包括该元素和该元素的
                                              值的儿子列表组成
H = [ 'H', [] ]                                # 构造叶子 H
I = [ 'I', [] ]                                # 构造叶子 I
K = [ 'K', [] ]                                # 构造叶子 K
E = [ 'E', [ G, H, I, K ] ]                    # 构造 E 节点
D = [ 'D', [] ]                                # 构造叶子 D
F = [ 'F', [] ]                                # 构造叶子 F
A = [ 'A', [ D, E ] ]                          # 构造 A 节点
B = [ 'B', [] ]                                # 构造叶子 B
C = [ 'C', [ F ] ]                              # 构造 C 节点
Root = [ 'Root', [ A, B, C ] ]                  # 构造树根
print(Root)
```

【代码说明】此处的代码很简单，就是使用嵌套的列表由底层向根逐渐构造列表树。

【运行效果】代码运行结果如下：

```
D:\lx\18>a18_3.py
['Root', [[['A', [[['D', []], ['E', [[['G', []], ['H', []], ['I', []], ['K', []]]]]], ['B', []], ['C', [[['F', []]]]]]]]]
```

18.2.2 实现二叉树类与遍历二叉树

二叉树是一类比较特殊的树，在二叉树中每个节点最多只有两个儿子，分别为左和右，如图 18.8 所示。相对于树而言，二叉树的构建和使用都要简单得多。

任何一棵树都可以通过变换转换成一棵二叉树。

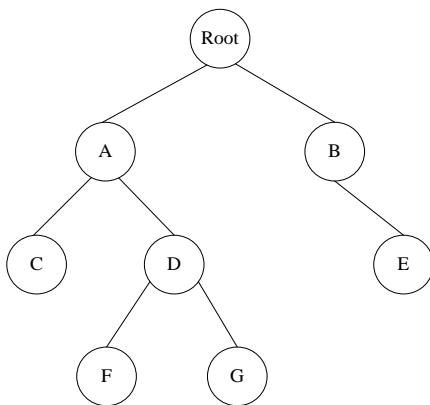


图 18.8 二叉树

在 Python 中, 二叉树的构建和树一样, 可以使用列表或者类的方式。由于二叉树中的节点具有确定的儿子数, 因此, 使用类的方式要更为简便。

【实例 18-4】 演示了用比较简单的方式生成了如图 18.8 所示的树, 代码如下:

```

# -*- coding:utf-8 -*-
#
class BTree:
    def __init__(self, value):
        self.left = None
        self.data = value
        self.right = None
    def insertLeft(self, value):
        self.left = BTree(value)
        return self.left
    def insertRight(self, value):
        self.right = BTree(value)
        return self.right
    def show(self):
        print(self.data)
if __name__ == '__main__':
    Root = BTree('Root')
    A = Root.insertLeft('A')
    C = A.insertLeft('C')
    D = A.insertRight('D')
    F = D.insertLeft('F')
    G = D.insertRight('G')
    B = Root.insertRight('B')
    E = B.insertRight('E')
    Root.show()
    Root.left.show()
    Root.right.show()
    A = Root.left
    A.left.show()
    Root.left.right.show()

```

二叉树节点
初始化函数
左儿子
节点值
右儿子
向左子树插入节点
向右子树插入节点
输出节点数据
根节点
向根节点中插入 A 节点
向 A 节点中插入 C 节点
向 A 节点中插入 D 节点
向 D 节点中插入 F 节点
向 D 节点中插入 G 节点
向根节点中插入 B 节点
向 B 节点中插入 E 节点
输出节点数据

【代码说明】 代码中首先定义了一个表示节点的类 `BTree`, 在该类构造方法中, 建立了三个实例变量, 分别用来保存其节点值、左子节点和右子节点, 然后通过实例化根节点不断从根开始构造该二叉树。

当创建好一棵二叉树后, 可以按照一定的顺序对树中所有的元素进行遍历。按照先左后右,



树的遍历方法有三种：先序遍历、中序遍历和后序遍历。

先序遍历的次序是：如果二叉树不为空，则访问根节点，然后访问左子树，最后访问右子树；否则，程序退出。

中序遍历的次序是：如果二叉树不为空，则先访问左子树，然后访问根节点，最后访问右子树；否则，程序退出。

后序遍历的次序是：如果二叉树不为空，则先访问左子树，然后访问右子树，最后访问根节点。

【实例 18-5】演示了使用三种遍历方式遍历如图 18.8 所示的树，代码如下：

```
# -*- coding:utf-8 -*-
#
class BTree:                                # 二叉树节点
    def __init__(self, value):                # 初始化函数
        self.left = None                     # 左儿子
        self.data = value                     # 节点值
        self.right = None                    # 右儿子
    def insertLeft(self, value):               # 向左子树插入节点
        self.left = BTree(value)
        return self.left
    def insertRight(self, value):              # 向右子树插入节点
        self.right = BTree(value)
        return self.right
    def show(self):                           # 输出节点数据
        print(self.data)
def preorder(node):                           # 先序遍历
    if node.data:
        node.show()
        if node.left:
            preorder(node.left)
        if node.right:
            preorder(node.right)
def inorder(node):                            # 中序遍历
    if node.data:
        if node.left:
            inorder(node.left)
        node.show()
        if node.right:
            inorder(node.right)
def postorder(node):                          # 后序遍历
    if node.data:
        if node.left:
            postorder(node.left)
        if node.right:
            postorder(node.right)
        node.show()
if __name__ == '__main__':
    Root = BTree('Root')                     # 构建树
    A = Root.insertLeft('A')
    C = A.insertLeft('C')
    D = A.insertRight('D')
    F = D.insertLeft('F')
    G = D.insertRight('G')
    B = Root.insertRight('B')
    E = B.insertRight('E')
    print('*****')
    print('Binary Tree Pre-Traversal')
    print('*****')
```

```

preorder(Root)                                # 对树进行先序遍历
print('*****')
print('Binary Tree In-Traversal')
print('*****')
inorder(Root)                                  # 对树进行中序遍历
print('*****')
print('Binary Tree Post-Traversal')
print('*****')
postorder(Root)                                # 对树进行后序遍历

```

【代码说明】代码中定义了与实例 18-4 相同的类，之后分别定义了先序遍历、中序遍历、后序遍历的函数，在遍历函数中采用了递归调用的方式来完成功能。

【运行效果】演示了使用三种遍历方式遍历如图 18.8 所示的树，其遍历结果如下：

运行 TreeTraversal.py 程序后输出如下：

```

*****
Binary Tree Pre-Traversal
*****
Root
A
C
D
F
G
B
E
*****
Binary Tree In-Traversal
*****
C
A
F
D
G
Root
B
E
*****
Binary Tree Post-Traversal
*****
C
F
G
D
A
E
B
Root

```

18.2.3 用字典构建与搜索图

图也是非线性的数据结构，由顶点和边组成。如果图中的顶点是有序的，那么图是有方向的，可称之为有向图，如图 18.9 所示；否则，图是无方向的，称之为无向图。在图中，由顶点组成的序列被称为路径。

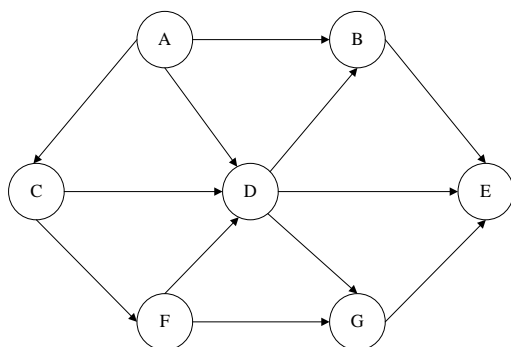


图 18.9 有向图

和树相比，图少了树明显的层次结构。在 Python 中，可以采用字典的方式来创建图，图中的每个元素是字典中的键，该元素所指向的是图中其他元素组成键的值。

同树一样，对于图来说，也可以对其进行遍历。除了遍历以外，可以在图中搜索所有的从一个顶点到另一个顶点的路径。图中的每一顶点可以看作一个城市，路径可以看作城市到城市之间的公路。因此，通过搜索所有的路径，可以找到一个顶点到另一个顶点间的最短路径，即城市到城市间的最短路线。

【实例 18-6】演示了使用字典的方式构建了如图 18.9 所示的有向图，并搜索图中的路径，代码如下：

```

# -*- coding:utf-8 -*-
#
def searchGraph(graph, start, end):                                # 搜索树
    results = []
    generatePath(graph, [start], end, results)                    # 生成路径
    results.sort(key=lambda x:len(x))                             # 按路径长短排序
    return results

def generatePath(graph, path, end, results):                      # 生成路径
    state = path[-1]
    if state == end:
        results.append(path)
    else:
        for arc in graph[state]:
            if arc not in path:
                generatePath(graph, path + [arc], end, results)

if __name__ == '__main__':
    Graph = {'A': ['B', 'C', 'D'],                               # 构建树
             'B': ['E'],
             'C': ['D', 'F'],
             'D': ['B', 'E', 'G'],
             'E': [],
             'F': ['D', 'G'],
             'G': ['E']}

    r = searchGraph(Graph, 'A', 'D')                             # 搜索 A 到 D 的所有路径
    print('*****')
    print('    path A to D')
    print('*****')
    for i in r:
        print(i)

    r = searchGraph(Graph, 'A', 'E')                             # 搜索 A 到 E 的所有路径
    print('*****')

```

```

print('    path A to E')
print('*****')
for i in r:
    print(i)
r = searchGraph(Graph, 'C','E')           # 搜索 C 到 E 的所有路径
print('*****')
print('    path C to E')
print('*****')
for i in r:
    print(i)

```

【代码说明】代码中采用字典来构造图，并定义了搜索图的函数 `generatePath()`，其中也使用了递归调用。

【运行效果】运行本例代码，将输出如下所示的内容：

```

*****
    path A to D
*****
['A', 'D']
['A', 'C', 'D']
['A', 'C', 'F', 'D']
*****
    path A to E
*****
['A', 'B', 'E']
['A', 'D', 'E']
['A', 'C', 'D', 'E']
['A', 'D', 'B', 'E']
['A', 'D', 'G', 'E']
['A', 'C', 'D', 'B', 'E']
['A', 'C', 'D', 'G', 'E']
['A', 'C', 'F', 'D', 'E']
['A', 'C', 'F', 'G', 'E']
['A', 'C', 'F', 'D', 'B', 'E']
['A', 'C', 'F', 'D', 'G', 'E']
*****
    path C to E
*****
['C', 'D', 'E']
['C', 'D', 'B', 'E']
['C', 'D', 'G', 'E']
['C', 'F', 'D', 'E']
['C', 'F', 'G', 'E']
['C', 'F', 'D', 'B', 'E']
['C', 'F', 'D', 'G', 'E']

```

18.3 查找与排序

查找和排序是最基本的算法，在很多程序中都会用到查找和排序。其实，在前边各章的例子中已多次使用 Python 的函数查找字符串中的子字符串。尽管 Python 提供的用于查找和排序的函数能够满足绝大多数需求，但还是有必要了解最基本的查找和排序算法，以便在有特殊需求的情况下，可以编写自己的查找、排序程序。

18.3.1 实现二分查找

基本的查找方法有顺序查找、二分查找和分块查找等。其中，顺序查找是最简单的查找方法，就是按数据排列的顺序依次查找，直到找到所查找的数据为止（可查看数据表都未找到的数据）。



二分查找是首先对要进行查找的数据进行排序，比如有按大小顺序排好的 9 个数字，如图 18.10 所示，如果要查找数字 5，首先与中间值 10 进行比较，5 小于 10，于是对序列的前半部分 1~9 进行查找。此时，中间值为 5，恰好为要找的数字，查找结束。

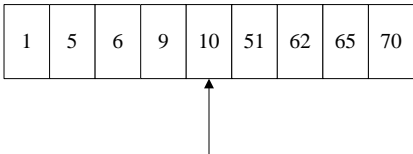


图 18.10 二分查找

分块查找，是介于顺序查找和二分查找之间的一种查找方法。使用分块查找时，首先对查找表建立一个索引表，再进行分块查找。建立索引表时，首先对查找表进行分块，要求“分块有序”，即块内关键字不一定有序，但分块之间有大小顺序。索引表是抽取各块中的最大关键字及其起始位置构成的，如图 18.11 所示。

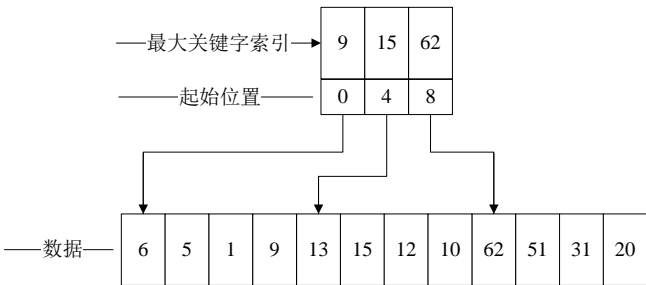


图 18.11 分块查找

分块查找分两步进行，首先查索引表，因为索引表是有序的，查找索引表时可以使用二分查找法进行。查找完索引表以后，就确定了要查找的数据所在的分块，然后在该分块中再进行顺序查找。

【实例 18-7】演示了对一个有序列表使用二分查找的实例，其代码如下：

```
# -*- coding:utf-8 -*-
#
def BinarySearch(l, key):                                     # 二分查找
    low = 0
    high = len(l) - 1
    i = 0
    while (low <= high):
        i = i + 1
        mid = (high + low) // 2
        if (l[mid] < key):
            low = mid + 1
        elif (l[mid] > key):
            high = mid - 1
        else:
            print('use %d time(s)' % i)
            return mid
    return -1
if __name__ == '__main__':
    l = [1, 5, 6, 9, 10, 51, 62, 65, 70]                     # 构造列表
    print(BinarySearch(l, 5))                                 # 在列表中查找
    print(BinarySearch(l, 10))
    print(BinarySearch(l, 65))
```

```
print(BinarySearch(1, 70))
```

【代码说明】代码很简单，仅定义了一个用于二分查找的函数 `BinarySearch()`，然后在主程序中调用它进行测试和查找。

【运行效果】运行程序后输出如下：

```
use 2 time(s)
1
use 1 time(s)
4
use 3 time(s)
7
use 4 time(s)
8
```

18.3.2 用二叉树排序

相对于查找来说，排序要复杂得多，排序的方法也较多，常用的排序方法有冒泡法排序、希尔排序、二叉树排序和快速排序等。其中二叉树排序是比较有意思的一种排序方法，而且也便于操作。二叉树排序的过程主要是二叉树的建立和遍历的过程。例如有一组数据“3, 5, 7, 20, 43, 2, 15, 30”，则二叉树的建立过程如下：

- (1) 首先将第一个数据 3 放入根节点；
- (2) 将数据 5 与根节点中的数据 3 比较，由于 5 大于 3，则将 5 放入 3 的右子树中；
- (3) 将数据 7 与根节点中的数据 3 比较，由于 7 大于 3，则应将 7 放入 3 的右子树中，由于 3 已经有右儿子 5，则将 7 与 5 进行比较，因为 7 大于 5，应将 7 放入 5 的右子树中；
- (4) 将数据 20 与根节点 3 进行比较，由于 20 大于 3，则应将 20 放入 3 的右子树，重复比较，最终将 20 放到 7 的右子树中；
- (5) 将数据 43 与树中的节点值进行比较，最终将其放入 20 的右子树中；
- (6) 将数据 2 与根节点 3 进行比较，由于 2 小于 3，则应将 2 放入 3 的左子树；
- (7) 同样对数据 15 和 30 进行处理，最终形成如图 18.12 所示的树。

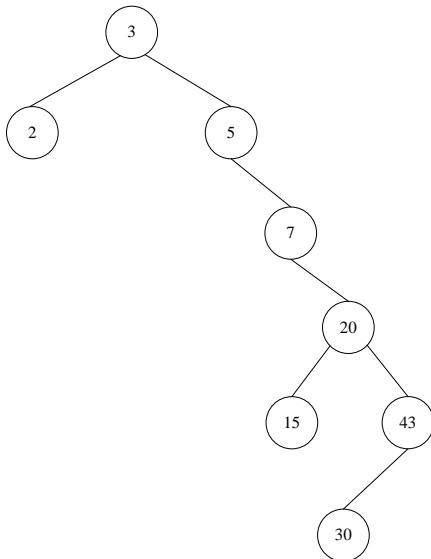


图 18.12 二叉树排序

当树创建好后，对树进行中序遍历，得到的遍历结果就是对数据从小到大的排序。如果要



从大到小进行排序，则可以先从右子树开始进行中序遍历。

【实例 18-8】演示了一个采用二叉树排序的方式对数据进行排序的实例，其代码如下：

```
# -*- coding:utf-8 -*-
#
class BTree:                                # 二叉树节点
    def __init__(self, value):                # 初始化函数
        self.left = None                     # 左儿子
        self.data = value                    # 节点值
        self.right = None                    # 右儿子

    def insertLeft(self, value):               # 向左子树插入节点
        self.left = BTree(value)
        return self.left

    def insertRight(self, value):              # 向右子树插入节点
        self.right = BTree(value)
        return self.right

    def show(self):                           # 输出节点数据
        print(self.data)

def inorder(node):                           # 中序遍历
    if node.data:
        if node.left:
            inorder(node.left)
        node.show()
        if node.right:
            inorder(node.right)

def rinorder(node):                          # 中序遍历,先遍历右子树
    if node.data:
        if node.right:
            rinorder(node.right)
        node.show()
        if node.left:
            rinorder(node.left)

def insert(node, value):
    if value > node.data:
        if node.right:
            insert(node.right, value)
        else:
            node.insertRight(value)
    else:
        if node.left:
            insert(node.left, value)
        else:
            node.insertLeft(value)

if __name__ == '__main__':
    l = [3, 5, 7, 20, 43, 2, 15, 30]
    Root = BTree(l[0])                        # 根节点
    node = Root
    for i in range(1, len(l)):
        insert(Root, l[i])
    print('*****')
    print('      从小到大')
    print('*****')
    inorder(Root)
```

```
print('*****')
print('        从大到小')
print('*****')
rinorder(Root)
```

【运行效果】运行该示例程序后，输出如下所示的排序结果：

```
*****
        从小到大
*****
2
3
5
7
15
20
30
43
*****
        从大到小
*****
43
30
20
15
7
5
3
2
```

18.4 小结

本章介绍了 Python 在处理基本数据结构方法的程序编写方法。由于 Python 设计良好的数据结构，通过列表就可以方便地完成表、栈、队列、树、图等数据结构的操作，因此，掌握了 Python 列表数据类型的使用，编写操作数据结构中的表、栈、队列、树、图的程序就比较简单了。最后，本章还介绍了用 Python 编写查找与排序的程序。

18.5 本章习题

一、简答题

1. 表、栈和队列有什么区别与联系？
2. 什么是树的遍历？它有哪些用途？
3. 除了使用字典，你还可以使用哪些 Python 提供的数据结构来构建图？
4. 二分查找的原理是什么，它有何优点？

二、实验题

1. 请使用堆栈来测试一段 HTML 代码中的 HTML 标签是否配对。
2. 用树来模拟一个文件系统，包括建立文件夹，存入指定名称的文件，显示当前文件系统中的所有文件夹和文件。
3. 编程实现生成一个具有 10000 个不同随机数的列表，并测试对比逐个查找与先排序后进行二分查找所使用的时间。

第 19 章 用 Pillow 库处理图片

Pillow 是 Python 2.x 时代比较流行的 Python Imaging Library（简称 Pillow）图像处理库的分支，见修复了一些 bug。Pillow 提供了对 Python 3 的支持，为 Python 3 解释器提供了图像处理的功能。和 Pillow 库一样提供了广泛的文件格式支持、高效的内部表示以及相当强大的图像处理功能。这个图像处理库的核心被设计成为能够快速访问几种基本像素类型表示的图像数据。它为通用图像处理工具提供了一个坚实基础，通过使用 Pillow 模块，可以使用 Python 3 对图片进行处理，例如，可用来对图片进行尺寸、格式、色彩、旋转等处理。

本章内容包括：

- Pillow 简介；
- 安装 Pillow；
- Pillow 图像基础；
- Image 模块基础；
- ImageChops 模块基础；
- ImageEnhance 模块基础；
- ImageFilter 模块基础；
- 使用 Pillow 转换图片格式；
- 使用 Pillow 生成缩略图；
- 使用 Pillow 为图片添加 Logo。

19.1 第三方 Pillow 库概述

由于 Pillow 不是 Python 自带的模块，因此需要用户自己安装。Pillow 是跨平台的，在各种系统下都可以使用 Pillow 的强大功能。



19.1.1 安装第三方 Pillow 库

与安装 Python 的其他第三方库的方法相同，可以到以下网址下载库的压缩包：

<https://pypi.python.org/pypi/Pillow/2.7.0>

<https://github.com/python-Pillow/Pillow>

解压后，在命令提示符下进入其目录，运行以下命令：

```
python setup.py install
```

如果你的计算机可以联网，直接运行以下命令就自动可以从互联网上下载安装：

```
pip install pillow
```

19.1.2 Pillow 库简介

Pillow 主要提供了以下对图片进行处理的模块。利用这些模块可以装载和保存多种格式文件，对图像进行缩放、裁剪、合成与变换，也可以像 Photoshop 一样专门处理通道，支持像素级操作、滤镜，支持对图像进行对比增强和统计分析，支持绘制文字和基本图形，甚至还支持动画。它包括的主要模块及其作用如表 19.1 所示。

表 19.1 Pillow的主要模块及其作用

模块名	作用
Image	Pillow 的主要模块
ImageChops	图片计算模块
ImageColor	颜色模块
ImageDraw	绘图模块
ImageEnhance	图片效果模块
ImageFile	图片文件存取模块
ImageFilter	图片过滤模块
ImageFont	字形模块, 用于绘图
ImageGrab	图片抓取模块
ImageOps	图片处理模块
ImagePath	路径队列模块
ImagePalette	图片调色板模块
ImageSequence	队列包装模块
ImageStat	图片属性模块
ImageTk	提供对 Tkinter 的支持
ImageWin	提供对 Windows 的支持
PSDraw	提供对 Postscript 的支持

19.1.3 Pillow 库处理图像基础

Pillow 库对图像的操作牵涉到图像处理的相关概念及表示如下所示。

- **pixels (像素)**: 构成图片的点单元;
- **size (尺寸)**: 图片的大小, 由两个元素的元组构成, 形式为水平像素数, 垂直像素数;
- **coordinates (坐标)**: 以左上角为 (0, 0) 的坐标系统, 形式为 (x,y);
- **angles (角度)**: 以 x 轴正方向为起点, 逆时针为正, 反之为负, 单位为度 (degree);
- **bboxes (边界框)**: 用来表示一个图像区域, 形式为 (x0,y0,x1,y1), 是以左上角点坐标 (x0,y0) 为起点至右正下角 (x1,y1) 为终点, 但不包含 x1 所在列和 y1 所在行的区域;
- **bands (通道)**: 独立的颜色通道, 它们具有相同的维数和深度。如在 RGB 模式下, 每个图片由三个通道 (RGB) 叠加而成;
- **mode (模式)**: 各种图片模式, 如表 19.2 所示;

表 19.2 Pillow支持图片模式及字符串表示

mode 模式	bands 通道数	意 义
"1"	1	黑白二值图像, 每像素用 0 或 1 共 1 位二进制代码表示
"L"	1	灰度图, 每像素用 8 位二进制代码表示
"P"	1	索引图, 每像素用 8 位二进制代码表示
"RGB"	3	24 位真彩图, 每像素用 3 个字节二进制代码表示
"RGBA"	4	"RGB" + 透明通道表示, 每像素用 4 个字节二进制代码表示
"CMYK"	4	印刷模式图像, 每像素用 4 个字节二进制代码表示
"YCbCr"	3	(彩色视频) 颜色隔离模式, 每像素用 3 个字节二进制代码表示
"LAB"	3	Lab 颜色空间, 每像素用 3 个字节二进制代码表示



续表

mode 模式	bands 通道数	意 义
“HSV”	3	每像素用 3 个字节二进制代码表示
“I”	1	整数形式表示像素，每像素用 4 个字节二进制代码表示
“F”	1	浮点数形式表示像素，每像素用 4 个字节二进制代码表示

- color (颜色): 可以为 32 位数值、元组或字符串。对于以上不同的图像格式，单通道图像像素值为 1 位、8 位、32 位等整数类型值；多通道图像则用像素元组来表示，即 (r,g,b)。此外，还可以使用以下风格的字符串型表示颜色。
 - CSS #FF0000 或 #F00;
 - RGB rgb(255,0,0 或 rgb(100%,0,0);
 - HSL hsl(120,30%,78%)。
- filters: 滤镜，多个像素映射到一个像素，以改变图像尺寸。目前主要有以下几种。
 - NEAREST 最近;
 - BILINEAR 双线性;
 - BICUBIC 双二次;
 - ANTIALIAS 平滑。
- fonts (字体): 可以支持 bitmap、OpenType、TrueType 等类型;
- format (格式): 文件格式，默认以文件名的扩展名指定，也可以单独指定，如 “PNG” “JPG” 等。



注意

以上各种概念（如图片模式等）在图片处理中很重要，要熟练处理图片，必须了解这些基础知识。

19.1.4 Image 模块中函数的使用

对于简单的图片处理，一般仅需使用 Image 模块，这个模块中包含的主要函数和使用方法如下：

```
open(file, mode)
```

其参数含义如下：

- file 要打开的图片文件；
- mode 可选参数，打开文件的方式，一般使用默认值 r 即可。

Image 模块中主要的函数是 open 函数，用于打开图片，操作结果是返回 Image 类的实例。当文件不存在时，会引发 IOError 错误：

```
new(mode, size, color=0)
```

其参数含义如下：

- mode 图片模式，如表 19.2 所示；
- size 图片尺寸，宽、高构成的元组；
- color 默认颜色（黑）。

```
blend(im1, im2, alpha)
```

其参数含义如下：

- im1 参与混合的图片 1；

- im2 参与混合的图片 2;
- alpha 混合透明度[0—1]。

实现的功能是将 im1、im2 两幅图片（尺寸相同）以一定的透明度混合，混合的方法是：

$(im1 \times (1 - \alpha) + im2 \times \alpha)$

当 alpha 为 0 时，得到 im1 原图片，当 alpha 为 1 时，得到 im2 原图片。

【实例 19-1】演示了用 blend()函数混合图像的实例，代码如下：

```
from PIL import Image
```

```
imga = Image.open('a.jpg') #打开图片 a.jpg
```

```
imgb = Image.open('b.jpg') #打开图片 b.jpg
```

```
Image.blend(imga, imgb, 0.5).show() #以 alpha 为 0.5 混合图片，并显示
```

【代码说明】代码仅有四行，但实现了两张图片以一定的透明度混合，并且混合后调用系统中的软件显示相应的图片。

【运行效果】将两幅图片混合后则如图 19.1 所示。



图 19.1 混合图片



注意

原始图片 a.jpg、b.jpg 分别如图 19.2、图 19.3 所示。



图 19.2 原图 a.jpg



图 19.3 原图 b.jpg

```
composite(im1, im2, mask)
```

其参数含义如下：

- im1 参与混合的图片 1；
- im2 参与混合的图片 2；
- mask 混合遮罩。

其功能是使用 mask 来混合 im1 和 im2，并且要求它们尺寸相同，作为 mask 的图片模式可以是“1”“L”“RGBA”。

【实例 19-2】演示了用 blend()函数中的遮罩功能混合图像的实例，代码如下：

```
from PIL import Image

imga = Image.open('a.jpg')      #打开图片 a.jpg
imgb = Image.open('b.jpg')      #打开图片 b.jpg
mask = Image.open('c.jpg')      #打开图片 c.jpg

Image.composite (imga,imgb, mask).show()    #以 c.jpg 为 mask 混合图片，并显示
```

【代码说明】代码仅有四行，但实现了用 mask 混合两张图片，并且混合后调用系统中的软件显示相应的图片。

【运行效果】将此两幅图片混合后则如图 19.4 所示。



图 19.4 应用遮罩功能后的合成图片

```
eval(image, fun)
```

其参数含义如下。

- im1 输入的图片；
- fun 对输入图片的每个像素应用此函数。

其功能是将输入图片的每个像素应用函数 fun 进行计算后返回，fun 函数只允许接收一个整数类型参数。如果一个图片含有多个通道，则每个通道都会应用这个函数。

【实例 19-3】演示了使用 eval() 函数对图片进行处理的实例，代码如下：

```
from PIL import Image

def div2(v):
    return v//2

imga = Image.open('a.jpg')

Image.eval(imga,div2).show()
```

#定义应用到每个像素的函数
#将像素值除以 2
#打开输入文件
#处理并显示图片

【代码说明】代码中定义了一个用于计算处理像素的函数 div2(), 然后打开一个输入图片，并调用 eval() 函数进行处理后输出。

【运行效果】对图片进行计算后效果如图 19.5 所示。



图 19.5 将像素值缩小一半后的图像

```
merge(mode, bands)
```

其参数含义如下：

- mode 输出文件的模式；
- bands 用于合成图片的所有通道列表、元组等序列。

其功能是将多个图形通道合成一幅图像，并返回该合成图像实例。

19.1.5 Image 模块中 Image 类的使用

上一节打开图片文件返回和引用图像的变量，实际上是 Image 模块中 Image 类对象的实例，运用该方法可以访问图片的属性和对图片进行变换操作。Image 类具有许多有用的方法和属性，如下所示。

主要属性有：

- Image.format 源图像格式；
- Image.mode 图像模式字符串；
- Image.size 图像尺寸。

主要方法有：

```
Image.convert(mode=None, matrix=None, dither=None, palette=0, colors=256)
```

其主要参数含义如下：

- mode 转换文件为此模式；
- matrix 转换使用的矩阵（4 或 16 元素的浮点数元组）；
- dither 设为 None 时，则所有非零值设置为 255（白色），也可以设置为参数 FLOYDSTEINBERG。

其功能是返回模式转换后的图像实例，目前支持的模式有“L”“RGB”“CMYK”，matrix



参数只支持“L”“RGB”。



注意 非原地修改，函数的返回值为修改后的图像。

彩色图片轮换为“L”模式时，应用以下公式进行转换：

$$L = R * 299/1000 + G * 587/1000 + B * 114/1000$$

此外，运用 `matrix` 参数可以转换出不同的、具有意想不到效果的图片：

`Image.transpose(method)`

变换图像后，返回变换的图像（非原地修改），方法有：

- `PIL.Image.FLIP_LEFT_RIGHT` 左右镜像；
- `PIL.Image.FLIP_TOP_BOTTOM` 上下镜像；
- `PIL.Image.ROTATE_90` 旋转 90 度；
- `PIL.Image.ROTATE_180` 旋转 180 度；
- `PIL.Image.ROTATE_270` 旋转 270 度；
- `PIL.Image.TRANSPOSE` 颠倒顺序。

`Image.paste(im, box=None, mask=None)`

其参数的意义如下：

- `im` 源图或像素值；
- `box` 为粘贴区域；
- `mask` 为遮罩。

功能是粘贴源至该图像，`box` 可以为以下三种情况：

- `(x1,y1)` 将源图像左上角对齐 `x1,y1` 点，其余超出被粘贴的图像的区域被抛弃；
- `(x1,y1,x2,y2)` 源图像与此区域必须一致；
- `None` 源图像与被粘贴的图像大小必须一致。

其他一些简单的方法如下：

```
Image.copy()
#复制图片，可用于处理或粘贴时需要持有源图片
vcrop(box=None)
#剪切图片中 box 所指区域
Image.filter(filter)
#对图片应用滤镜（可用的滤镜在 Imagefilter 模块中）
Image.getbands()
#获取图像每个通道的名称列表，RGB 图像则返回：['R','G','B']
Image.getextrema()
#获取图像最大、最小像素点值
Image.getpixel(xy)
#获取像素点值
Image.histogram(mask=None, extrema=None)
#获取图像直方图，返回像素计数的列表
Image.offset(xoffset, yoffset=None)
#偏移图像
Image.point(function)
#使用函数修改图像每个像素
Image.putalpha(alpha)
#添加或替换图像的 alpha 层
Image.putdata(data, scale=1.0, offset=0.0)
#将序列像素值复制到图片上，使用 scale 与 offset 参数时意义如下：pixel=value*scale+offset.
Image.resize(size, resample=0)
#重新设置图片尺寸
```

```

Image.rotate(angle, resample=0, expand=0)
#旋转图片
Image.save(fp, format=None, **params)
#保存图片
Image.show(title=None, command=None)
#显示图片
Image.split()
#将图片分隔为多个通道列表
Image.thumbnail(size, resample=3)
#生成缩略图, 原地修改
Image.transform(size, method, data=None, resample=0, fill=1)
#变换图像
Image.verify()
#校验文件是否损坏
Image.close()
#关闭文件

```

【实例 19-4】 演示了 Image 模块的一些函数的使用, 代码如下:

```

from PIL import Image

imga = Image.open('a.jpg')
print('图像格式: ',imga.format)
print('图像模式: ',imga.mode)
print('图像尺寸: ',imga.size)
print('图像通道列表: ',imga.getbands())
# print('统计直方图列表: ',imga.histogram())
imgb = imga.copy()                                #复制图像 imga
imgb.thumbnail((224,168))                         #生成图像的缩略图
imgb.show()                                       #显示图像其尺寸为指定的大小(224,168)

imgc = imga.copy()                                #复制图像 imga
region = imgb.crop((50,50,120,120))              #裁剪出一块区域
imgc.paste(region,(130,130))                     #将裁剪出区域粘贴到图像 imgc
imgc.show()                                       #显示图像

img_output = Image.new('RGB',(448,168))          #创建一个新图像(用于放置对比的图像)
img_output.paste(imgb,(0,0))                     #粘贴未被处理的图像
img_output.show()                                #显示
b = imgb.convert('CMYK')                         #得到一幅被转换为 CMYK 的模式图
img_output.paste(b,(224,0))                     #粘贴转换后的图像
img_output.show()                               #显示粘贴后的图像

flip = b.transpose(Image.FLIP_LEFT_RIGHT)        #得到一幅左右镜像的图像
img_output.paste(flip,(224,0))                  #粘贴镜像的图像
img_output.show()                               #显示对比图

b = imgb.convert('L')                           #转换图像为灰度图
img_output.paste(b,(224,0))                     #粘贴灰度图
img_output.show()                               #显示对比图

b = imgb.offset(30)                              #对图像进行移位
img_output.paste(b,(224,0))                     #粘贴移位后的图像
img_output.show()                               #显示对比图

b = imgb.rotate(45)                              #旋转图像
img_output.paste(b,(224,0))                     #粘贴旋转后的图像
img_output.show()                               #显示对比图

```



```
chnls = imgb.split() #分离图像通道
b = Image.merge('RGB',chnls[::-1]) #合并 RB 互换后的通道
img_output.paste(b,(224,0)) #粘贴合并后的图像
img_output.show() #显示对比图

from PIL import ImageFilter
b = imgb.filter(ImageFilter.GaussianBlur) #对图片运用滤镜（高斯模糊）
img_output.paste(b,(224,0)) #粘贴模糊后的图像
img_output.show() #显示对比图

r,g,b = chnls
r_after = r.point(lambda i:i if i<100 else 255) #处理 R 通道图像中的每个像素
b = Image.merge("RGB",(r_after,g,b)) #合并变化 R 通道后的图像
img_output.paste(b,(224,0)) #粘贴 R 通道变换后的图像
img_output.show() #显示对比图
```

【代码说明】代码中首先导入 Pillow 库，然后用 Image 模块中的 open()函数打开当前目录下的图像文件“a.jpg”，并返回图像的实例。之后分别应用 Image 模块中的函数和 Image 模块中 Image 类的方法处理它，并输出其对比图。

【运行效果】运行该实例代码后，控制台输出如图 19.6 所示，输出了图像文件中的“a.jpg”的图像格式、图像模式、图像尺寸和图像通道列表。其他处理后的图像输出如图 19.7～19.16 所示。



图 19.6 控制台输出



图 19.7 将粘贴裁剪得到的图像粘贴至原来的图像中

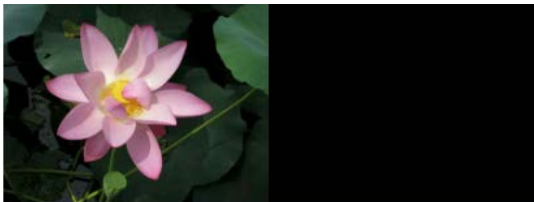


图 19.8 新建的用于对比处理效果的图像（原图在左）



图 19.9 转换为 CMYK 后的对比图 (原图在左)



图 19.10 水平镜像图片 (原图在左)

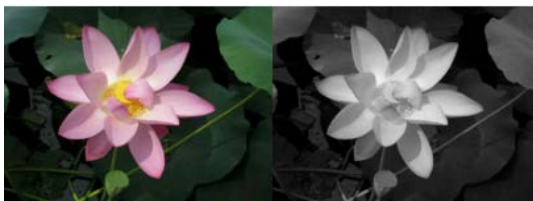


图 19.11 转换为灰度图 (原图在左)



图 19.12 对图像移位 (原图在左)



图 19.13 旋转图像 (原图在左)



图 19.14 RG 通道互换图像 (原图在左)



图 19.15 高斯模糊图像（原图在左）



图 19.16 R 通道处理得到图像（原图在左）

19.1.6 使用 ImageChops 模块进行图片合成

该模块包含多种用计算通道中像素值的方式来进行图片合成的函数，其主要用于制作特效、合成图片等。这个模块包含的图像处理的函数很多，这里仅举几个例子说明其用法，更详细的函数及其使用可以参考官方网站。



1. 相加

```
ImageChops.add(image1, image2, scale=1.0, offset=0)
```

合成后的图像的每个像素值是两幅图像对应像素值依据以下公式进行计算的像素值形成的图片：

```
out = ((image1 + image2) / scale + offset)
```

2. 减去

```
ImageChops.subtract(image1, image2, scale=1.0, offset=0)
```

合成后的图像的每个像素值是两幅图像对应像素值依据以下公式进行计算的像素值形成的图片：

```
out = ((image1 - image2) / scale + offset)
```

3. 变暗

```
ImageChops.darker(image1, image2)
```

计算后的图像像素取两张图像中对应像素的较小值，所以合成时两幅图像的对应位置的暗部得到保留，而亮部则被除去，像素的计算公式如下：

```
out = min(image1, image2)
```

4. 变亮

```
ImageChops.lighter(image1, image2)
```

该方法与 `darker()` 函数相反，计算后得到的图像是两幅图像对应位置的亮部。像素值的计算公式如下：

```
out = max(image1, image2)
```

5. 正片叠底

```
ImageChops.multiply(image1, image2)
```

合成的图像的效果类似两张画在透明的描图纸上叠放在一起观看的效果。其对应像素值的计算公式如下：

```
out = image1 * image2 / MAX
```

6. 屏幕

```
ImageChops.screen(image1, image2)
```

合成图像的效果就像是两张幻灯片用两台投影机同时投影到一个屏幕上的效果。其对应像素值的计算公式为：

```
out = image1 * image2 / MAX
```

7. 反相

```
ImageChops.invert(image)
```

将用 255 减去一幅图像的每个像素值，从而得到原来图像的反相。换句话说，其表现为“底片”性质的图像。像素值的计算公式如下：

```
out = MAX - image
```

8. 比较

```
ImageChops.difference(image1, image2)
```

将两幅图像的像素值对应相减后得到的图像，对应像素值相同的则为黑色。它可以用来寻找图像之间的差异，就像流行的“请你来找茬”图像游戏。像素值的计算公式如下：

```
out = abs(image1 - image2)
```

9. 灰度填充

```
ImageChops.constant(image, value)
```

用所给的灰度等级来填充各像素，用来生成给定的灰度值图像。

【实例 19-5】演示了用以上多个函数合成图片效果的例子，其代码如下：

```
from PIL import Image
from PIL import ImageChops

imga = Image.open('a.jpg')
imgb = Image.open('b.jpg')

ImageChops.add(imga, imgb, 1, 0).show()
ImageChops.subtract(imga, imgb, 1, 0).show()
ImageChops.darker(imga, imgb).show()
ImageChops.lighter(imga, imgb).show()
ImageChops.multiply(imga, imgb).show()
ImageChops.screen(imga, imgb).show()
ImageChops.invert(imga).show()
ImageChops.difference(imga, imga).show()
```

【代码说明】代码很简单，先导入使用的模块后打开用来合成的两幅图像，之后调用 ImageChops 模块中的函数来合成图像。

【运行效果】运行后得到的图像如图 19.17~19.23 所示，而最后产生的图像为纯黑色，因为比较的是同一幅图像。



图 19.17 相加



图 19.18 相减

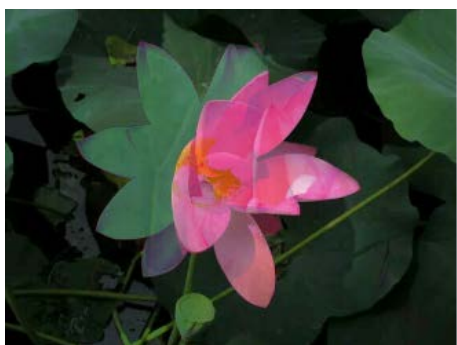


图 19.19 变暗



图 19.20 变亮

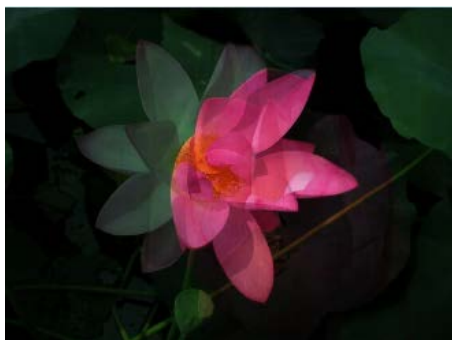


图 19.21 正片叠底



图 19.22 屏幕

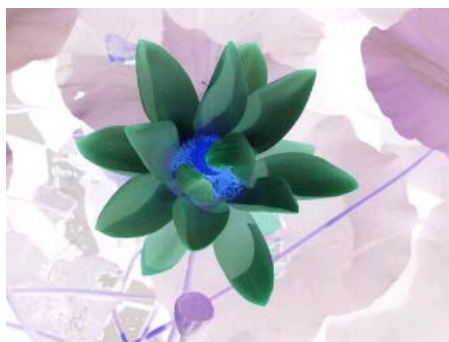


图 19.23 反相

19.1.7 使用 ImageEnhance 模块增强图像效果

该模块中函数的主要作用是增强图像的效果，主要可以用来调整图像的色彩、对比度、亮度、清晰度等，感觉上和调整电视机的显示参数一样。其主要的函数如下：

- `ImageEnhance.Color(image)` 图像色彩平衡调整；
- `ImageEnhance.Contrast(image)` 图像对比度调整；
- `ImageEnhance.Brightness(image)` 图像亮度调整；
- `ImageEnhance.Sharpness(image)` 图像清晰度调整。

其使用步骤如下：

- (1) 打开图像；
- (2) 创建对应的增强调整器；
- (3) 调用调整器输出函数，将按指定增强系数（小于 1 表示减弱，大于 1 表示增强，等于 1 表示原图不变）调整后输出图像。

【实例 19-6】演示了使用图像增强模块的实例，代码如下：

```
from PIL import Image
from PIL import ImageEnhance

imga = Image.open('a.jpg')           #打开图像文件

w,h = imga.size                       #获取图像的宽、高
img_output = Image.new('RGB',(2*w,h)) #建立用于对比显示的新文件
img_output.paste(imga,(0,0))          #粘贴原图

nhc = ImageEnhance.Color(imga)        #建立图像色彩调整器
nhb = ImageEnhance.Brightness(imga)    #建立图像亮度调整器
for nh in [nhc,nhb]:                  #使用内嵌循环程序输出调整后的图像
    for ratio in [0.6,1.8]:            #使用减弱和增强两种系数
        b = nh.enhance(ratio)
        img_output.paste(b,(w,0))      #粘贴调整后的图像
        img_output.show()               #显示对比图像
```

【代码说明】代码首先导入所需的 Pillow 库，然后分别建立两种调整器（色彩、亮度），最后在两层循环中输出调整后的图像对比。

【运行效果】运行此代码输出的 4 幅图像如图 19.24~19.27 所示。



图 19.24 色彩减弱



图 19.25 色彩增强



图 19.26 亮度减弱



图 19.27 亮度增强

19.1.8 使用 ImageFilter 模块的滤镜

ImageFilter 模块提供了滤镜功能，它也可以被用来建立图像特效，或以此效果作为中间结果进一步处理。该模块中提供了一些预定义的和供自定义的过滤器。

预定义的近十个过滤器中主要有：

- BLUR 模糊滤镜；
- FIND_EDGES 查找边缘；
- SHARPEN 锐化；
- EDGE_ENHANCE 边缘增强。

当然还有七八个可以有自定义值的滤镜，主要有：

- 高斯模糊 ImageFilter.GaussianBlur(radius=2)；
- USM 锐化 ImageFilter.UnsharpMask(radius=2, percent=150, threshold=3)；
- 中值滤波 ImageFilter.MedianFilter(size=3)；
- 最小值滤波 ImageFilter.MinFilter(size=3)；
- 模式滤波 ImageFilter.ModeFilter(size=3)。

它们的使用方法也很简单，就是把滤镜实例作为参数提供给前文介绍的 Image 类的方法 filter()，即可返回具有特效的图像。

【实例 19-7】 演示了应用以上几种滤镜生成的对比图的实例，代码如下：

```
from PIL import Image
from PIL import ImageFilter

imga = Image.open('a.jpg')           #打开文件

w,h = imga.size                       #获取图像宽、高
img_output = Image.new('RGB',(2*w,h)) #新建图像文件
img_output.paste(imga,(0,0))          #粘贴原图

fltrs = []                            #建立空滤镜列表
fltrs.append(ImageFilter.EDGE_ENHANCE) #边缘强化滤镜
fltrs.append(ImageFilter.FIND_EDGES)   #查找边缘滤镜
fltrs.append(ImageFilter.GaussianBlur(4)) #高斯模糊滤镜
```

```

for fltr in fltrs:                                #循环使用以上三种滤镜
    r = imga.filter(fltr)                        #应用滤镜
    img_output.paste(r,(w,0))                    #波幅应用后的图像
    img_output.show()                            #显示对比图像

```

【代码说明】代码中首先导入了相关的库，之后建立应用滤镜的列表，并用 for 遍历使用并输出其对比图。

【运行效果】运行以上代码得到的对比效果图如图 19.28～19.30 所示。



图 19.28 边缘增强

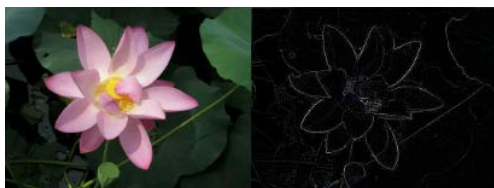


图 19.29 查找边缘



图 19.30 高斯模糊

19.1.9 使用 ImageDraw 模块画图

ImageDraw 模块提供了 2D 绘图功能，可以通过创建图片来绘图，也可以在原有的图片上进行绘图，以达到修饰图片或对图片进行注释的目的。

使用 ImageDraw 模块画图的基本步骤为：

- (1) 新建或打开一个文件；
- (2) 建立一个 ImageDraw.Draw 对象，并且提供指向文件的参数；
- (3) 引用上一步建立的对象方法进行绘图；
- (4) 可以保存或直接输出图像。

ImageDraw.Draw 对象的主要绘图方法有：

arc(xy, start, end, fill=None)	#绘制圆弧
ellipse(xy, fill=None, outline=None)	#绘制椭圆
line(xy, fill=None, width=0)	#绘制直线
point(xy, fill=None)	#绘制点
rectangle(xy, fill=None, outline=None)	#绘制矩形
text(xy, text, fill=None, font=None, anchor=None)	#绘制字符串
setfill(fill)	#设置默认填充颜色



```
setfont(font) #设置默认字体
```

要在图像上绘制字符串，可以选择使用的字体。这就需要引用 `ImageFont` 模块，这里简要说明一下：

```
load(filename) #返回可直接使用的字体对象
truetype(font=None,size=10,encoding='') #以给定的字体、字号创建字体对象
```

【实例 19-8】演示了一个用 `ImageDraw` 模块绘制的图形的实例，代码如下：

```
from PIL import Image
from PIL import ImageDraw

a = Image.new('RGB',(200,200),'white') #创建一个背景为白色的图像
drw = ImageDraw.Draw(a) #创建绘图对象
drw.rectangle((50,50,150,150),outline='red') #绘制矩形
drw.text((60,60),'My First Draw',fill='green') #绘制文本
a.show()
```

【代码说明】代码中使用了 `ImageDraw` 对象在新建的图像中使用其方法 `rectangle()` 和 `text()` 分别绘制了一个矩形和一个字符串。

【运行效果】运行结果输出的图像如图 19.31 所示。



图 19.31 用 `ImageDraw` 绘图

19.2 使用 Pillow 库处理图片举例

Pillow 提供了非常实用的函数，很多情况下仅需使用一两个 Pillow 函数或者方法，就可以完成对图片的处理，如调整图片大小、转换图片格式等。配合 Python 的灵活性，使用 Pillow 可以创建一些非常实用的图片处理程序。本节主要介绍使用 Pillow 结合 `tkinter` 库建立图形化批量图片处理工具。

19.2.1 图片格式转换

使用 Pillow 转换图片格式，主要使用 Pillow 的 `Image` 模块。首先使用 `Image.open` 函数打开文件，然后将文件保存为所需要的格式即可。`Image` 可以根据文件的扩展名自动选择文件保存的格式，因此不需要设置文件格式。

【实例 19-9】演示了批量图片文件格式转换的 GUI 程序，代码如下：

```
# -*- coding:utf-8 -*-
#
import os # 导入模块
from Pillow import Image
import tkinter
import tkinter.filedialog
import tkinter.messagebox
class Window: # 创建窗口
```

```

def __init__(self):
    self.root = root = tkinter.Tk()
    label = tkinter.Label(root, text = '选择目录')
    label.place(x = 5, y = 5)
    self.entry = tkinter.Entry(root)
    self.entry.place(x=60, y = 5)
    self.buttonBrowser = tkinter.Button(root, text = '浏览',
        command = self.Browser)
    self.buttonBrowser.place(x=210, y = 5)
    frameF = tkinter.Frame(root)
    frameF.place(x = 5, y = 30)
    frameT = tkinter.Frame(root)
    frameT.place(x = 100, y = 30)
    self.fImage = tkinter.StringVar()
    self.tImage = tkinter.StringVar()
    self.status = tkinter.StringVar()
    self.fImage.set('.bmp')
    self.tImage.set('.bmp')
    labelFrom = tkinter.Label(frameF, text = 'From')
    labelFrom.pack(anchor='w')
    labelTo = tkinter.Label(frameT, text = 'To')
    labelTo.pack(anchor='w')
    frBmp = tkinter.Radiobutton(frameF, variable = self.fImage,
        value = '.bmp', text = 'BMP' )
    frBmp.pack(anchor='w')
    frJpg = tkinter.Radiobutton(frameF, variable = self.fImage,
        value = '.jpg', text = 'JPG' )
    frJpg.pack(anchor='w')
    frGif = tkinter.Radiobutton(frameF, variable = self.fImage,
        value = '.gif', text = 'GIF' )
    frGif.pack(anchor='w')
    frPng = tkinter.Radiobutton(frameF, variable = self.fImage,
        value = '.png', text = 'PNG' )
    frPng.pack(anchor='w')
    trBmp = tkinter.Radiobutton(frameT, variable = self.tImage,
        value = '.bmp', text = 'BMP' )
    trBmp.pack(anchor='w')
    trJpg = tkinter.Radiobutton(frameT, variable = self.tImage,
        value = '.jpg', text = 'JPG' )
    trJpg.pack(anchor='w')
    trGif = tkinter.Radiobutton(frameT, variable = self.tImage,
        value = '.gif', text = 'GIF' )
    trGif.pack(anchor='w')
    trPng = tkinter.Radiobutton(frameT, variable = self.tImage,
        value = '.png', text = 'PNG' )
    trPng.pack(anchor='w')
    self.buttonConv = tkinter.Button(root, text = '转换',
        command = self.Conv)
    self.buttonConv.place(x=80, y = 160)
    self.labelStatus = tkinter.Label(root, textvariable=self.status)
    self.labelStatus.place(x=50, y = 195)
def MainLoop(self):
    self.root.minsize(250,220)
    self.root.maxsize(250,220)
    self.root.mainloop()
def Browser(self):
    directory = tkinter.filedialog.askdirectory(title='Python')
    if directory:
        self.entry.delete(0, tkinter.END)
        self.entry.insert(tkinter.END, directory)
def Conv(self):
    n = 0

```

创建组件

生成关联变量

进入消息循环

浏览目录

转换文件格式



```

t = self.tImage.get()
f = self.fImage.get()
path = self.entry.get()
if path == '':
    tkinter.messagebox.showerror('Python tkinter','请输入路径')
    return
filenames = os.listdir(path)
os.mkdir(path + '/' + t[-3:])
for filename in filenames:
    if filename[-4:] == f:
        Image.open(path + '/' + filename).save(path +
            '/' + t[-3:] + '/' + filename[:-4] + t)
        n = n + 1
self.status.set('成功转换%d张图片' % n)
window = Window()
window.mainloop()

```

【代码说明】代码的主要部分为自定义的一个主类，类的构造函数中生成 GUI 窗口组件，并在窗口中添加相应的 GUI 组件供用户选择使用，主类的 `Browser()` 方法用来选择转换的图片文件的目录，`Conv()` 方法用来遍历当前目录中的所选类型的图片文件并转换保存。

【运行效果】程序运行后，将显示如图 19.32 所示窗口，单击“浏览”按钮选择一个保存源图片的目录，接着在下方选择从一种图片格式转换为另一种图片格式，最后单击“转换”按钮即可批量转换图片的格式。转换操作完成后，在指定目录下新建一个目录，用来保存转换后的图片。



图 19.32 转换文件格式

19.2.2 批量生成缩略图

生成缩略图也是常用的图片处理方式之一，使用 Pillow 生成缩略图可以使用 `Image` 的 `resize` 函数，因为使用 `resize` 函数可以重新指定图片的大小。

【实例 19-10】演示了程序使用 Pillow 模块批量生成图片的缩略图，代码如下：

```

# -*- coding:utf-8 -*-
#
import os                                     # 导入模块
from Pillow import Image
import tkinter
import tkinter.filedialog
import tkinter.messagebox

class Window:
    def __init__(self):
        self.root = root = tkinter.Tk()      # 创建窗口
        self.Image = tkinter.StringVar()

```

```

self.status = tkinter.StringVar()
self.mstatus = tkinter.IntVar()
self.fstatus = tkinter.IntVar()
self.Image.set('bmp')
self.mstatus.set(0)
self.fstatus.set(0)
label = tkinter.Label(root, text = '输入百分比')
label.place(x = 5, y = 5)
self.entryNew = tkinter.Entry(root)
self.entryNew.place(x = 70, y = 5)
self.checkM = tkinter.Checkbutton(root, text = '批量转换',
                                   command = self.OnCheckM,
                                   variable = self.mstatus,
                                   onvalue = 1,
                                   offvalue = 0)
self.checkM.place(x = 5, y = 30)
label = tkinter.Label(root, text = '选择文件')
label.place(x = 5, y = 55)
self.entryFile = tkinter.Entry(root)
self.entryFile.place(x=60, y = 55)
self.buttonBrowserFile = tkinter.Button(root, text = '浏览',
                                         command = self.BrowserFile)
self.buttonBrowserFile.place(x=200, y = 55)
label = tkinter.Label(root, text = '选择目录')
label.place(x = 5, y = 80)
self.entryDir = tkinter.Entry(root,
                               state = tkinter.DISABLED)
self.entryDir.place(x=60, y = 80)
self.buttonBrowserDir = tkinter.Button(root, text = '浏览',
                                         command = self.BrowserDir,
                                         state = tkinter.DISABLED)
self.buttonBrowserDir.place(x=200, y = 80)

self.checkF = tkinter.Checkbutton(root, text = '改变文件格式',
                                   command = self.OnCheckF,
                                   variable = self.fstatus,
                                   onvalue = 1,
                                   offvalue = 0)
self.checkF.place(x = 5, y = 110)
frame = tkinter.Frame(root)
frame.place(x = 10, y = 130)
labelTo = tkinter.Label(frame, text = 'To')
labelTo.pack(anchor='w')
self.rBmp = tkinter.Radiobutton(frame, variable = self.Image,
                                value = 'bmp', text = 'BMP',
                                state = tkinter.DISABLED)
self.rBmp.pack(anchor='w')
self.rJpg = tkinter.Radiobutton(frame, variable = self.Image,
                                value = 'jpg', text = 'JPG',
                                state = tkinter.DISABLED)
self.rJpg.pack(anchor='w')
self.rGif = tkinter.Radiobutton(frame, variable = self.Image,
                                value = 'gif', text = 'GIF',
                                state = tkinter.DISABLED)
self.rGif.pack(anchor='w')
self.rPng = tkinter.Radiobutton(frame, variable = self.Image,
                                value = 'png', text = 'PNG',
                                state = tkinter.DISABLED)
self.rPng.pack(anchor='w')
self.buttonConv = tkinter.Button(root, text = '转换',
                                  command = self.Conv)
self.buttonConv.place(x=100, y = 175)

```



```

        self.labelStatus = tkinter.Label(root, textvariable=self.status)
        self.labelStatus.place(x=80, y=205)
    def MainLoop(self):                                # 进入消息循环
        self.root.minsize(250, 270)
        self.root.maxsize(250, 250)
        self.root.mainloop()
    def BrowserDir(self):                                # 选择路径
        directory = tkinter.filedialog.askdirectory(title='Python')
        if directory:
            self.entryDir.delete(0, tkinter.END)
            self.entryDir.insert(tkinter.END, directory)
    def BrowserFile(self):                                # 选择文件
        file = tkinter.filedialog.askopenfilename(title='Python Music Player',
            filetypes=[('JPG', '*.jpg'), ('BMP', '*.bmp'),
                ('GIF', '*.gif'), ('PNG', '*.png')])
        if file:
            self.entryFile.delete(0, tkinter.END)
            self.entryFile.insert(tkinter.END, file)
    def OnCheckM(self):                                # 设置组件状态
        if not self.mstatus.get():
            self.entryDir.config(state=tkinter.DISABLED)
            self.entryFile.config(state=tkinter.NORMAL)
            self.buttonBrowserDir.config(state=tkinter.DISABLED)
            self.buttonBrowserFile.config(state=tkinter.NORMAL)
        else:
            self.entryDir.config(state=tkinter.NORMAL)
            self.entryFile.config(state=tkinter.DISABLED)
            self.buttonBrowserDir.config(state=tkinter.NORMAL)
            self.buttonBrowserFile.config(state=tkinter.DISABLED)
    def OnCheckF(self):                                # 设置组件状态
        if not self.fstatus.get():
            self.rBmp.config(state=tkinter.DISABLED)
            self.rJpg.config(state=tkinter.DISABLED)
            self.rGif.config(state=tkinter.DISABLED)
            self.rPng.config(state=tkinter.DISABLED)
        else:
            self.rBmp.config(state=tkinter.NORMAL)
            self.rJpg.config(state=tkinter.NORMAL)
            self.rGif.config(state=tkinter.NORMAL)
            self.rPng.config(state=tkinter.NORMAL)
    def Conv(self):                                    # 转换图片
        n = 0
        if self.mstatus.get():
            path = self.entryDir.get()
            if path == '':
                tkinter.messagebox.showerror('Python tkinter', '请输入路径')
                return
            filenames = os.listdir(path)
            if self.fstatus.get():
                f = self.Image.get()
                for filename in filenames:
                    if filename[-3:] in ('bmp', 'jpg', 'gif', 'png'):
                        self.make(path + '/' + filename, f)
                        n = n + 1
            else:
                for filename in filenames:
                    if filename[-3:] in ('bmp', 'jpg', 'gif', 'png'):
                        self.make(path + '/' + filename)
                        n = n + 1
        else:
            file = self.entryFile.get()
            if file == '':

```

```

        tkinter.messagebox.showerror('Python tkinter','请选择文件')
        return
    if self.fstatus.get():
        f = self.Image.get()
        self.make(file, f)
        n = n + 1
    else:
        self.make(file)
        n = n + 1
    self.status.set('成功转换%d 图片' % n)
def make(self, file, format = None):
    im = Image.open(file)
    mode = im.mode
    if mode not in ('L', 'RGB'):
        im = im.convert('RGB')
    width, height = im.size
    s = self.entryNew.get()
    if s == '':
        tkMessageBox.showerror('Python tkinter','请输入百分比')
        return
    else:
        n = int(s)
        nwidth = int(width * n / 100)
        nheight = int(height * n / 100)
        thumb = im.resize((nwidth,nheight), Image.ANTIALIAS)
        if format:
            thumb.save(file[:len(file) - 4] + '_thumb.' + format)
        else:
            thumb.save(file[:len(file) - 4] + '_thumb' + file[-4:])
window = Window()
window.MainLoop()

```

生成缩略图

【代码说明】该程序的代码和实例 19-9 基本结构相同，这里不做解释了。

【运行效果】程序运行后将显示如图 19.33 所示的窗口，输入缩略图的百分比，接着选择一个文件或目录进行转换，单击“转换”按钮，即可为选择的图片或选择目录中所有的图片生成缩略图。生成的缩略图与原图保存同一个目录中，缩略图的文件名由原文件名加上“_thumb.jpg”组成。

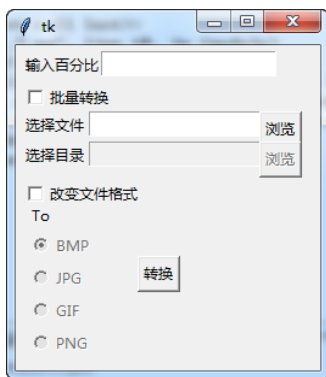


图 19.33 生成的缩略图

19.2.3 为图片添加 Logo

使用 Pillow 为图片添加 Logo，主要使用 Image 的 paste 函数。利用 paste 函数可以往图片中粘贴其他的图片。



【实例 19-11】 演示的程序代码，就是使用 Pillow 模块为图片批量添加 Logo：

```
# -*- coding:utf-8 -*-
#
import os                                     # 导入模块
from Pillow import Image
import tkinter
import tkinter.filedialog
import tkinter.messagebox

class Window:
    def __init__(self):
        self.root = root = tkinter.Tk()      # 创建窗口
        self.Image = tkinter.StringVar()
        self.status = tkinter.StringVar()
        self.mstatus = tkinter.IntVar()
        self.fstatus = tkinter.IntVar()
        self.pstatus = tkinter.IntVar()
        self.Image.set('bmp')
        self.mstatus.set(0)
        self.fstatus.set(0)
        self.pstatus.set(0)
        label = tkinter.Label(root, text = 'Logo')
        label.place(x = 5, y = 5)
        self.entryLogo = tkinter.Entry(root)
        self.entryLogo.place(x = 50, y = 5)
        self.buttonBrowserLogo = tkinter.Button(root, text = '浏览',
            command = self.BrowserLogo)
        self.buttonBrowserLogo.place(x = 200, y = 5)
        self.checkM = tkinter.Checkbutton(root, text = '批量转换',
            command = self.OnCheckM,
            variable = self.mstatus,
            onvalue = 1,
            offvalue = 0)
        self.checkM.place(x = 5, y = 30)
        label = tkinter.Label(root, text = '选择文件')
        label.place(x = 5, y = 55)
        self.entryFile = tkinter.Entry(root)
        self.entryFile.place(x = 60, y = 55)
        self.buttonBrowserFile = tkinter.Button(root, text = '浏览',
            command = self.BrowserFile)
        self.buttonBrowserFile.place(x=200, y = 55)
        label = tkinter.Label(root, text = '选择目录')
        label.place(x = 5, y = 80)
        self.entryDir = tkinter.Entry(root,
            state = tkinter.DISABLED)
        self.entryDir.place(x=60, y = 80)
        self.buttonBrowserDir = tkinter.Button(root, text = '浏览',
            command = self.BrowserDir,
            state = tkinter.DISABLED)
        self.buttonBrowserDir.place(x=200, y = 80)

        self.checkF = tkinter.Checkbutton(root, text = '改变文件格式',
            command = self.OnCheckF,
            variable = self.fstatus,
            onvalue = 1,
            offvalue = 0)
        self.checkF.place(x = 5, y = 110)
        frame = tkinter.Frame(root)
        frame.place(x = 10, y = 130)
        labelTo = tkinter.Label(frame, text = '格式')
```

```

labelTo.pack(anchor='w')
self.rBmp = tkinter.Radiobutton(frame, variable = self.Image,
                                value = 'bmp', text = 'BMP',
                                state = tkinter.DISABLED)
self.rBmp.pack(anchor='w')
self.rJpg = tkinter.Radiobutton(frame, variable = self.Image,
                                value = 'jpg', text = 'JPG',
                                state = tkinter.DISABLED)
self.rJpg.pack(anchor='w')
self.rGif = tkinter.Radiobutton(frame, variable = self.Image,
                                value = 'gif', text = 'GIF',
                                state = tkinter.DISABLED)
self.rGif.pack(anchor='w')
self.rPng = tkinter.Radiobutton(frame, variable = self.Image,
                                value = 'png', text = 'PNG',
                                state = tkinter.DISABLED)
self.rPng.pack(anchor='w')
pframe = tkinter.Frame(root)
pframe.place(x = 70, y = 130)
labelPos = tkinter.Label(pframe, text = '位置')
labelPos.pack(anchor = 'w')
self.rLT = tkinter.Radiobutton(pframe, variable = self.pstatus,
                                value = 0, text = '左上角')
self.rLT.pack(anchor = 'w')
self.rRT = tkinter.Radiobutton(pframe, variable = self.pstatus,
                                value = 1, text = '右上角')
self.rRT.pack(anchor = 'w')
self.rLB = tkinter.Radiobutton(pframe, variable = self.pstatus,
                                value = 2, text = '左下角')
self.rLB.pack(anchor = 'w')
self.rRB = tkinter.Radiobutton(pframe, variable = self.pstatus,
                                value = 3, text = '右下角')
self.rRB.pack(anchor = 'w')
self.buttonAdd = tkinter.Button(root, text = '添加',
                                command = self.Add)
self.buttonAdd.place(x=180, y = 175)
self.labelStatus = tkinter.Label(root, textvariable=self.status)
self.labelStatus.place(x=150, y = 205)

def MainLoop(self):                                     # 进入消息循环
    self.root.minsize(250,270)
    self.root.maxsize(250,270)
    self.root.mainloop()

def BrowserLogo(self):
    file = tkinter.filedialog.askopenfilename(title = 'Python Music Player',
        filetypes=[('JPG', '*.jpg'), ('BMP', '*.bmp'),
                    ('GIF', '*.gif'), ('PNG', '*.png')])
    if file:
        self.entryLogo.delete(0, tkinter.END)
        self.entryLogo.insert(tkinter.END, file)

def BrowserDir(self):                                   # 选择路径
    directory = tkinter.filedialog.askdirectory(title='Python')
    if directory:
        self.entryDir.delete(0, tkinter.END)
        self.entryDir.insert(tkinter.END, directory)

def BrowserFile(self):                                  # 选择文件
    file = tkinter.filedialog.askopenfilename(title = 'Python Music Player',
        filetypes=[('JPG', '*.jpg'), ('BMP', '*.bmp'),
                    ('GIF', '*.gif'), ('PNG', '*.png')])
    if file:
        self.entryFile.delete(0, tkinter.END)
        self.entryFile.insert(tkinter.END, file)

def OnCheckM(self):                                     # 设置组件状态

```



```

if not self.mstatus.get():
    self.entryDir.config(state = tkinter.DISABLED)
    self.entryFile.config(state = tkinter.NORMAL)
    self.buttonBrowserDir.config(state = tkinter.DISABLED)
    self.buttonBrowserFile.config(state = tkinter.NORMAL)
else:
    self.entryDir.config(state = tkinter.NORMAL)
    self.entryFile.config(state = tkinter.DISABLED)
    self.buttonBrowserDir.config(state = tkinter.NORMAL)
    self.buttonBrowserFile.config(state = tkinter.DISABLED)
def OnCheckF(self):                                # 设置组件状态
    if not self.fstatus.get():
        self.rBmp.config(state = tkinter.DISABLED)
        self.rJpg.config(state = tkinter.DISABLED)
        self.rGif.config(state = tkinter.DISABLED)
        self.rPng.config(state = tkinter.DISABLED)
    else:
        self.rBmp.config(state = tkinter.NORMAL)
        self.rJpg.config(state = tkinter.NORMAL)
        self.rGif.config(state = tkinter.NORMAL)
        self.rPng.config(state = tkinter.NORMAL)
def Add(self):                                     # 处理图片
    n = 0
    if self.mstatus.get():
        path = self.entryDir.get()
        if path == '':
            tkinter.messagebox.showerror('Python tkinter', '请输入路径')
            return
        filenames = os.listdir(path)
        if self.fstatus.get():
            f = self.Image.get()
            for filename in filenames:
                if filename[-3:] in ('bmp', 'jpg', 'gif', 'png'):
                    self.addlogo(path + '/' + filename, f)
                    n = n + 1
        else:
            for filename in filenames:
                if filename[-3:] in ('bmp', 'jpg', 'gif', 'png'):
                    self.addlogo(path + '/' + filename)
                    n = n + 1
    else:
        file = self.entryFile.get()
        if file == '':
            tkinter.messagebox.showerror('Python tkinter', '请选择文件')
            return
        if self.fstatus.get():
            f = self.Image.get()
            self.addlogo(file, f)
            n = n + 1
        else:
            self.addlogo(file)
            n = n + 1
    self.status.set('成功添加%d张图片' % n)
def addlogo(self, file, format = None):            # 向图片添加 Logo
    logo = self.entryLogo.get()
    if logo == '':
        tkinter.messagebox.showerror('Python tkinter', '请选择 Logo')
        return
    im = Image.open(file)
    lo = Image.open(logo)
    imwidth = im.size[0]
    imheight = im.size[1]

```

```

lowidth = lo.size[0]
loheight = lo.size[1]
pos = self.pstatus.get()
if pos == 0:
    left = 0
    top = 0
    right = lowidth
    bottom = loheight
elif pos == 1:
    left = imwidth - lowidth
    top = 0
    right = imwidth
    bottom = loheight
elif pos == 2:
    left = 0
    top = imheight - loheight
    right = lowidth
    bottom = imheight
else:
    left = imwidth - lowidth
    top = imheight - loheight
    right = imwidth
    bottom = imheight
im.paste(lo, (left, top, right, bottom))
if format:
    im.save(file[:len(file) - 4] + '_logo.' + format)
else:
    im.save(file[:len(file) - 4] + '_logo' + file[-4:])
window = Window()
window.MainLoop()

```

【代码说明】其代码结构与前例相同。

【运行效果】程序运行后将显示如图 19.34 所示的窗口，选择一个作为 Logo 的图片，接着选择需要添加 Logo 的图片，然后在窗口下方选择 Logo 添加的位置，最后单击“添加”按钮，即可将 Logo 图片添加到图片的指定位置。



图 19.34 图片添加 Logo

19.3 小结

本章介绍了一个 Python 处理图片的第三方模块 Pillow，Pillow 模块的功能非常强大，可对图片进行各种操作。在本章中首先介绍了 Pillow 模块的下载和安装。然后介绍了 Pillow 模块的处理图像的基本概念、Image 模块、ImageChops 模块、ImageEnhance 模块、magefilter 模块等模块的基本使用方法。最后本章介绍了使用 Pillow 处理图片的几个案例：使用 Pillow 转换图片



格式、生成缩略图、为图片添加 Logo。此外，Pillow 模块中还包括其他一些模块，可以参考相关资料。通过学习本章的内容，你应掌握使用 Pillow 第三方模块进行图片处理的相关方法。

19.4 本章习题

一、简答题

1. Pillow 库中常用的图像的模式主要有哪些？
2. Pillow 库中颜色有哪三种表示方式？
3. Pillow 库主要有哪些模块，其主要功能是什么？
4. 二分查找的原理是什么，它有何优点？

二、实验题

现有一批照片，大小为 300×200 像素，为了洗印，需要将其依照文件名（序号+姓名命名的）排序，将每 6 张放入同一底版（大小为 680×700 像素）上，每张照片下方应印有序号和姓名。请应用 pillow 库来自动完成这一要求。

第 3 篇 Python 编程实战

第 20 章 案例 1 做一个 Windows 上的 360 工具

Python 的语法简洁而清晰，具有丰富和强大的类库及第三方库。它能够很轻松地将各种语言模块联结在一起，所以被称为“胶水”语言。当然，Python 也能够方便快捷地编写一些常用的工具程序，而用其他程序设计语言需要编写很复杂的代码来完成的功能，通过 Python 的类库，可能只需要几行代码就能完成同样的功能。

利用 Python 的这种快速开发特性，在日常计算机维护方面，Python 就大有用武之地。可以利用 Python 语言编写各种系统维护的程序，甚至还可以用它来开发远程维护或监控计算机系统的程序。本章将介绍一些典型的应用场景和实际案例。

本章内容包括：

- 创建 GUI；
- 迭代目录；
- 扫描垃圾文件；
- 多线程加速；
- 删除垃圾文件；
- 搜索大文件；
- 搜索文件名称。

20.1 案例背景



Windows 用户都有这样的体会，计算机使用一段时间后，感觉速度越来越慢，不但启动时速度变慢，启动后的运行速度也越来越慢。这是因为 Windows 操作系统在运行过程中会不断地产生垃圾文件，安装、卸载和使用应用程序，也同样会产生垃圾文件。

这些垃圾文件随着计算机的使用越来越多，如果得不到及时的清理，将会影响到计算机系统的运行速度。

既然知道是垃圾文件造成计算机速度变慢，那么解决方法也就有了，就是将这些垃圾文件及时清理掉。

清理垃圾文件的方式有多种：

- 逐个查看盘符、目录内是否存在垃圾文件，若有，则删除；
- 借助一些工具软件，通过这些软件可以轻松地扫描、删除垃圾文件；
- 利用所学的 Python 知识，自己编写清理垃圾文件的程序。

这几种方法各有优缺点：

- 第一种方法是人工地分辨垃圾文件，可以很自由地决定哪些是垃圾文件，但费时费力，效率很低；
- 第二种方法省时省力，其缺点是只能清理该软件认识的垃圾文件，无法辨识一些由特



殊软件产生的垃圾文件;

- 第三种方法是由 Python 程序去代替人工辨识垃圾文件, 当有特殊的文件类别需要清理时, 可以通过修改 Python 程序快速地删除这些文件。因此, 这种方式有第一种方法的灵活性, 且效率又有很大提高。当然, 刚开始编写的清理程序功能不是很完善, 需要用户在使用过程中不断提出新需求, 逐步完善其功能。

本章的案例主要考虑以下几方面, 要求通过编写 Python 程序解决这些问题:

- (1) 扫描垃圾文件, 统计垃圾文件数量及大小;
- (2) 删除垃圾文件;
- (3) 扫描计算机中的大文件, 并列出具体的文件;
- (4) 查找匹配部分文字名字符的文件。

本章将编写 Python 程序来解决这些问题。当然, 计算机维护还有很多功能, 参照本章编写的 Python 程序, 你也可以自己编写代码来解决问题。这样, 随着需求不断地增多, 最终就可得到一个比较完善的维护计算机的 Python 程序。

20.2 从创建图形化界面开始

根据上节分析的需求, 本章的案例需要完成多项功能, 可以将每一项功能单独编写一个 Python 程序, 在使用时分别执行这些程序即可。为了方便使用, 最好将这些功能集成在一起, 并使用图形化界面。这样, 通过单击鼠标就可执行相应的功能。

20.2.1 创建基本图形化工作界面

在编写创建 GUI 的程序之前, 要先对需求进行分析, 划分出功能模块, 如图 20.1 所示, 是一种功能模块划分方式。在这个图中, “搜索” 和 “清理” 分别完成搜索文件和清理垃圾文件的功能。另外增加了一个 “系统”, 主要用来提供退出的功能。

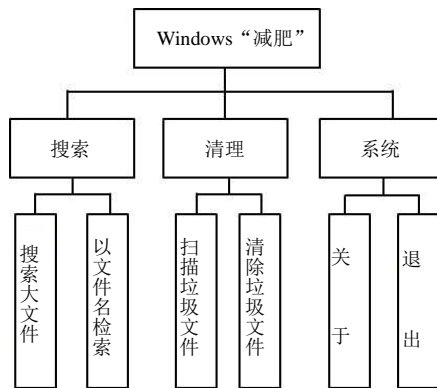


图 20.1 功能模块划分

从图 20.1 所示功能模块划分可看出, PytOptimize 需要完成六项工作, 功能不算太多, 可以考虑在 GUI 界面中创建六个按钮, 每个按钮连接一个功能。但是, 这种通过按钮来执行功能的方式不方便以后进行扩展, 如果功能扩展到几十个, 显然就不适合使用按钮了。对于功能项很多的程序, 最好的选择就是使用菜单。因此, 本例也使用菜单来驱动。



注意

在项目实施前, 一般都要事先对程序所要完成的功能进行规划或设计, 以便于划分模块或代码, 使代码结构清晰, 也容易实施代码的编写。

编写以下 Python 程序，通过使用 tkinter 模块创建用户界面：

```
#coding:utf-8
#file: findfat1.py

import tkinter
import tkinter.messagebox

class Window:
    def __init__(self):
        self.root = tkinter.Tk()

        #创建菜单
        menu = tkinter.Menu(self.root)

        #创建“系统”子菜单
        submenu = tkinter.Menu(menu, tearoff=0)
        submenu.add_command(label="关于...")
        submenu.add_separator()
        submenu.add_command(label="退出")
        menu.add_cascade(label="系统", menu=submenu)

        #创建“清理”子菜单
        submenu = tkinter.Menu(menu, tearoff=0)
        submenu.add_command(label="扫描垃圾文件")
        submenu.add_command(label="删除垃圾文件")
        menu.add_cascade(label="清理", menu=submenu)

        #创建“查找”子菜单
        submenu = tkinter.Menu(menu, tearoff=0)
        submenu.add_command(label="搜索大文件")
        submenu.add_separator()
        submenu.add_command(label="按名称搜索文件")
        menu.add_cascade(label="搜索", menu=submenu)

        self.root.config(menu=menu)

        #创建标签，用于显示状态信息
        self.progress = tkinter.Label(self.root, anchor = tkinter.W,
                                     text = '状态', bitmap = 'hourglass', compound = 'left')
        self.progress.place(x=10, y=370, width = 480, height = 15)

        #创建文本框，显示文件列表
        self.flist = tkinter.Text(self.root)
        self.flist.place(x=10, y = 10, width = 480, height = 350)

        #为文本框添加垂直滚动条
        self.vscroll = tkinter.Scrollbar(self.flist)
        self.vscroll.pack(side = 'right', fill = 'y')
        self.flist['yscrollcommand'] = self.vscroll.set
        self.vscroll['command'] = self.flist.yview

    def MainLoop(self):
        self.root.title("Findfat")
        self.root.minsize(500,400)
        self.root.maxsize(500,400)
        self.root.mainloop()

if __name__ == "__main__" :
    window = Window()
    window.MainLoop()
```




运行以上程序，将显示如图 20.2 所示的窗口。在窗口上方显示了三个下拉菜单，分别对应图 20.1 所示的模块。菜单下方是一个列表框，用来显示最终的处理结果，列表框右侧添加了一个垂直滚动条，当列表框中的内容太多时，可通过这个滚动条快速移动。在窗口的最下方还加了一个标签，用来显示当前正在扫描处理的文件。

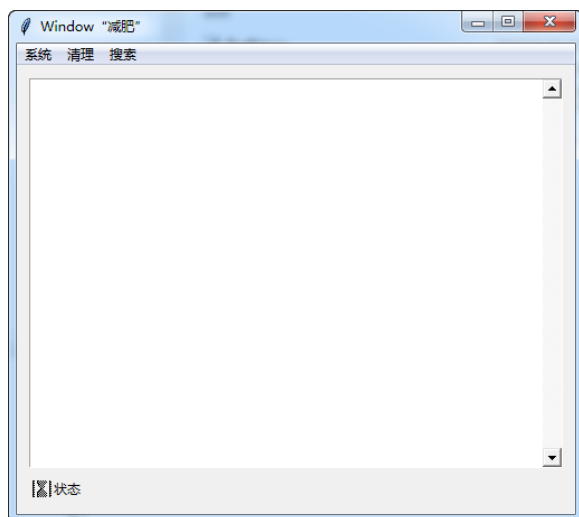


图 20.2 Findfat 窗口

20.2.2 响应菜单事件

在如图 20.2 所示窗口中单击选择“系统”→“退出”菜单命令，窗口并不会被关闭。这是因为还没有为菜单创建事件程序，因此，这里只有单击窗口右上角的关闭按钮来关闭窗口了。

接下来就为菜单创建事件程序。将 Findfat1.py 另存为 Findfat2.py，然后进行修改，将每个子菜单项都添加上 command 设置，修改后的代码如下：

```
#coding:utf-8
#file: findfat2.py

import tkinter
import tkinter.messagebox

class Window:
    def __init__(self):
        self.root = tkinter.Tk()

        #创建菜单
        menu = tkinter.Menu(self.root)

        #创建“系统”子菜单
        submenu = tkinter.Menu(menu, tearoff=0)
        submenu.add_command(label="关于...", command = self.MenuAbout)
        submenu.add_separator()
        submenu.add_command(label="退出", command = self.MenuExit)
        menu.add_cascade(label="系统", menu=submenu)

        #创建“清理”子菜单
        submenu = tkinter.Menu(menu, tearoff=0)
        submenu.add_command(label="扫描垃圾文件", command = self.MenuScanRubbish)
        submenu.add_command(label="删除垃圾文件", command = self.MenuDelRubbish)
```

```

menu.add_cascade(label="清理", menu=submenu)

#创建“查找”子菜单
submenu = tkinter.Menu(menu, tearoff=0)
submenu.add_command(label="搜索大文件", command = self.MenuScanBigFile)
submenu.add_separator()
submenu.add_command(label="按名称搜索文件", command = self.MenuSearchFile)
menu.add_cascade(label="搜索", menu=submenu)

self.root.config(menu=menu)
(以下部分省略)

```

在上面的程序中，为每个子菜单添加了 `command`。

这时还不能运行修改后的程序，运行时将显示如下的错误提示：

```

Traceback (most recent call last):
  File "Findfat2.py", line 58, in <module>
    window = Window()
  File "Findfat2.py", line 16, in __init__
    submenu.add_command(label="关于...",command = self.MenuAbout)
AttributeError: 'Window' object has no attribute 'MenuAbout'

```

以上错误提示的意思是“Window 对象没有 MenuAbout 属性”，是指还没有为 MenuAbout 编写相应的程序。对于“关于”菜单项，只需弹出一个对话框显示当前系统的一些提示信息。接着上面的程序，在 Window 类中编写以下函数：

```

# “关于”菜单
def MenuAbout(self):
    tkinter.messagebox.showinfo("Findfat",
        "这是使用 Python 编写的 Windows 优化程序。\\n 欢迎使用并提出宝贵意见！")

```

编写好 MenuAbout 函数后，再次运行程序，又会提示没有编写 MenuExit 函数（“退出”菜单项），继续添加以下程序：

```

#"退出"菜单
def MenuExit(self):
    self.root.quit();

```

类似地，还需对“清理”和“搜索”这两个下拉菜单中的菜单项编写相应的响应函数。由于还没有编写实现其功能的程序，这里就暂时编写一个替代程序（例如，当选择该菜单命令时，就显示一个对话框），到后面完成相应的功能程序时，再替换就行了。



一般来说，这种“暂时替代程序”可以让主程序暂时能够正常运行，便于调试代码。

具体程序如下：

```

#"扫描垃圾文件"菜单
def MenuScanRubbish(self):
    result = tkinter.messagebox.askquestion("Findfat", "扫描垃圾文件将需要较长的时间，是否继续?")
    if result == 'no':
        return
    tkinter.messagebox.showinfo("Findfat", "马上开始删除垃圾文件！")

#"删除垃圾文件"菜单
def MenuDelRubbish(self):
    result = tkinter.messagebox.askquestion("Findfat", "删除垃圾文件将需要较长的时间，是否继续?")
    if result == 'no':

```



```
        return
    tkinter.messagebox.showinfo("Findfat", "马上开始删除垃圾文件!")

    # "搜索大文件"菜单
    def MenuScanBigFile(self):
        result = tkinter.messagebox.askquestion("Findfat", "扫描大文件将需要较长的时间, 是否继续?")
        if result == 'no':
            return
        tkinter.messagebox.showinfo("Findfat", "马上开始扫描大文件!")

    # "按名称搜索文件"菜单
    def MenuSearchFile(self):
        result = tkinter.messagebox.askquestion("Findfat", "按名称搜索文件将需要较长的时间, 是否继续?")
        if result == 'no':
            return
        tkinter.messagebox.showinfo("Findfat", "马上开始按名称搜索文件!")
```

经过以上修改, 程序又能正常运行了。运行后将显示一个窗口, 选择菜单“系统”→“关于”命令, 将弹出一个信息提示窗口, 如图 20.3 所示。当然, 选择其他菜单命令也会弹出相应的信息提示窗口。现在, 选择“系统”→“退出”命令, 就可以关闭窗口了。

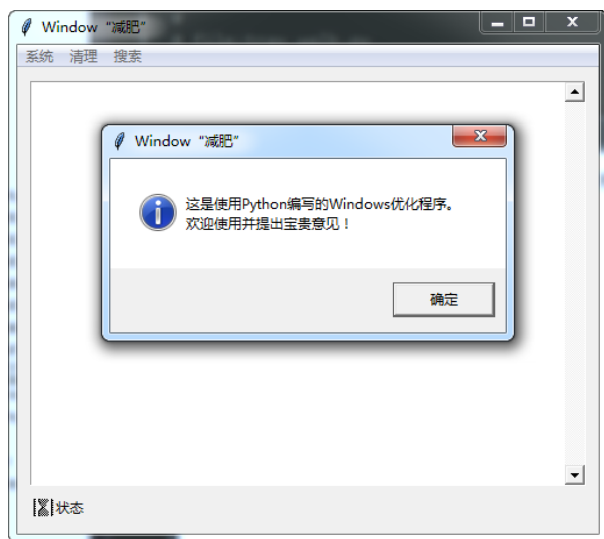


图 20.3 Findfat 窗口

20.3 清理垃圾文件

GUI 界面制作完成后, 接下来就该编写程序来实现相应的功能了。首先来实现清理垃圾文件的功能, 在“清理”下拉菜单中有两个子菜单项, “扫描垃圾文件”和“删除垃圾文件”。要完成这两项功能, 都需要找到垃圾文件所在位置, 由于计算机中垃圾文件的分布是随机的, 在各子目录中都有可能存在垃圾文件, 因此, 首先就需要编写程序对目录进行遍历。

20.3.1 迭代目录

要遍历目录, 有其他程序设计语言经验的读者首先就可以想到使用递归算法来进行遍历。例如, 以下程序就可遍历“C:\python32”目录, 将该目录及其子目录中的文件名逐个输

出。

```
# coding:utf-8
#
# file:traverse.py

import os,os.path

def traverse(pathname):
    for item in os.listdir(pathname):
        fullitem = os.path.join(pathname,item)
        print(fullitem)
        if os.path.isdir(fullitem):           #判断是否为目录
            traverse(fullitem)

traverse("d:/python34")
```

在上面的程序中，首先使用 `os` 模块中的 `listdir` 方法获取传入参数（目录）中的所有文件和子目录，然后循环处理。具体的处理过程是：首先输出当前项（文件或目录）的名称（使用 `join` 方法将父目录名和当前项连接起来），然后判断当前项为目录，则递归调用 `traverse` 函数处理下层子目录。有关递归算法的用法可参考算法相关的书籍。

当然，在以上程序中将函数 `traverse` 的参数换成其他目录，就可以遍历相应目录下的文件了。另外，在上面的遍历程序中，只是输出文件名，也可以将这里的输出操作换成其他操作。

在 Python 中，`os` 模块提供了一个名为 `walk` 的函数，这个函数就可以完成递归的功能。该函数将返回一个元组（`root,dirs,files`），其中的 `root` 表示当前目录，`dirs` 是当前目录下的所有子目录，而 `files` 则表示当前目录下的所有文件。下面的程序可完成对指定目录的遍历，并输出文件名（与 `traverse.py` 程序的功能相同）：

```
# coding:utf-8
#
# file:trav_walk.py

import os,os.path

def trav_walk(pathname):
    for root,dirs,files in os.walk(pathname):
        for fil in files:
            fname=os.path.abspath(os.path.join(root,fil))
            print(fname)

trav_walk("d:/python34")
```

在上面的程序中，只对返回元组中的 `files` 进行了循环输出，对下级子目录的遍历不用编写程序，由 `walk` 函数内部自己处理。

20.3.2 扫描垃圾文件

学会编写程序遍历目录后，再回到案例中来，接下来就可编写程序，遍历目录找出所有的垃圾文件，并进行统计。

在这里，先对垃圾文件进行定义。最简单的方法就是通过文件的扩展名进行界定，例如扩展名为 `tmp`、`bak`、`old`、`wbk`、`xlk`、`_mp`、`log`、`gid`、`chk`、`syd`、`$$$`、`@@@`、`~*`等的文件都是垃圾文件（当然，这个名章可以扩充）。因此，在遍历目录时就可判断文件扩展名是否为列表中的内容，若是，则该文件就是垃圾文件。

将上节中编写的 `Findfat2.py` 另存为 `Findfat3.py`，然后进行修改。

首先在程序开始处编写以下内容，导入 `os` 和 `os.path` 模块（处理文件需要使用），然后将



定义为垃圾文件的扩展名保存到一个全局变量 `rubbishExt` 列表中:

```
#coding:utf-8
#file: Findfat3.py

import tkinter
import tkinter.messagebox
import os,os.path

rubbishExt=['.tmp','.bak','.old','.wbk','.xlk','.mp','.log','.gid','.chk','.syd',
'.$$$','.@@@','.~*']
```

对于 `rubbishExt` 列表可进行扩展(注意,这里将 `log` 定义为垃圾文件,很多日志文件的扩展名为 `log`,如果调试或分析系统时需要用到,则不要将其定义为垃圾文件进行删除)。

接着,在 `Windows` 类中编写以下函数,进行垃圾文件扫描:

```
#扫描垃圾文件
def ScanRubbish(self):
    global rubbishExt
    total = 0
    filesize = 0
    for root,dirs,files in os.walk("c:/"):
        try:
            for fil in files:
                filesplit = os.path.splitext(fil)
                if filesplit[1] == '': #若文件无扩展名
                    continue
                try:
                    if rubbishExt.index(filesplit[1]) >=0: #扩展名在垃圾文
                        #扩展名列表中
                        fname = os.path.join(os.path.abspath(root),fil)
                        filesize += os.path.getsize(fname)
                        if total % 20 == 0:
                            self.flist.delete(0.0,tkinter.END)
                            self.flist.insert(tkinter.END,fname + '\n')
                            l = len(fname)
                            if l>60:
                                self.progress['text'] = fname[:30] + '...' +
                                fname[l-30:l]
                            else:
                                self.progress['text'] = fname
                            total += 1 #计数
                        except ValueError:
                            pass
                    except Exception as e:
                        print(e)
                        pass
                self.progress['text'] = "找到 %s 个垃圾文件,共占用 %.2f M 磁盘空间" %
                (total,filesize/1024/1024)
```

最后,修改函数 `MenuScanRubbish`,只需在原有程序下方增加一行调用 `ScanRubbish` 函数,修改后的 `MenuScanRubbish` 函数如下:

```
#"扫描垃圾文件"菜单
def MenuScanRubbish(self):
    result = tkinter.messagebox.askquestion("Findfat","扫描垃圾文件将需要较长的
    时间,是否继续?")
    if result == 'no':
        return
    tkinter.messagebox.showinfo("Findfat","马上开始扫描垃圾文件!")
    self.ScanRubbish()
```

编写好以上程序后,运行 Findfat3.py 将出现应用程序窗口,选择菜单“清理”→“扫描垃圾文件”命令后会发现一个问题,这时窗口无法响应操作了(如不能移动窗口),如图 20.4 所示,经过一段时间之后,标题栏中出现“无响应”提示。

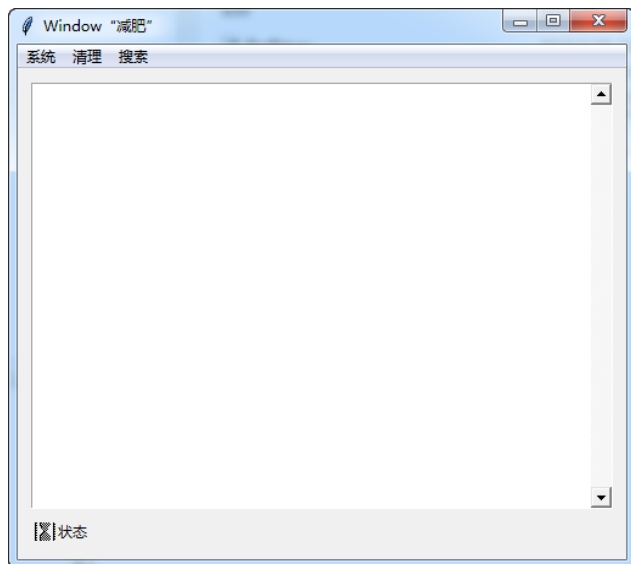


图 20.4 无响应

20.3.3 多线程加速

出现这个问题的原因是什么呢?

在 Python 程序遍历目录时需要用很长的时间,可能是几分钟,也可能是几十分钟甚至几个小时。在执行遍历程序时,窗口就无法响应其他操作。因此,就出现了前面的问题。

那么该怎么办?这时就应该使用 Python 的多线程操作了,将耗时的操作用另一个后台线程去进行,则前台 GUI 界面仍然可响应用户操作。

在使用多线程时需要导入 threading 模块,继续修改 Findfat.py 程序,在前面添加导入 threading 模块的程序,修改内容如下:

```
#coding:utf-8
#file: Findfat3.py
```

```
import tkinter
import tkinter.messagebox
import os,os.path
import threading
```

然后修改 MenuScanRubbish 函数,添加以创建线程和执行线程的两行程序,具体如下:

```
#"扫描垃圾文件"菜单
def MenuScanRubbish(self):
    result = tkinter.messagebox.askquestion("Findfat","扫描垃圾文件将需要较长的
时间,是否继续?")
    if result == 'no':
        return
    tkinter.messagebox.showinfo("Findfat","马上开始扫描垃圾文件!")
    #self.ScanRubbish()
    t=threading.Thread(target=self.ScanRubbish)           #创建线程
    t.start()                                             #开始线程
```



再次运行 Findfat.py 程序，在窗口中选择菜单“清理”→“扫描垃圾文件”命令，在窗口界面中就可以看到扫描的过程，文本框中将显示垃圾文件名，如图 20.5 所示，扫描完成后将在下方状态标签中显示找到的垃圾文件数量和占用的磁盘空间。

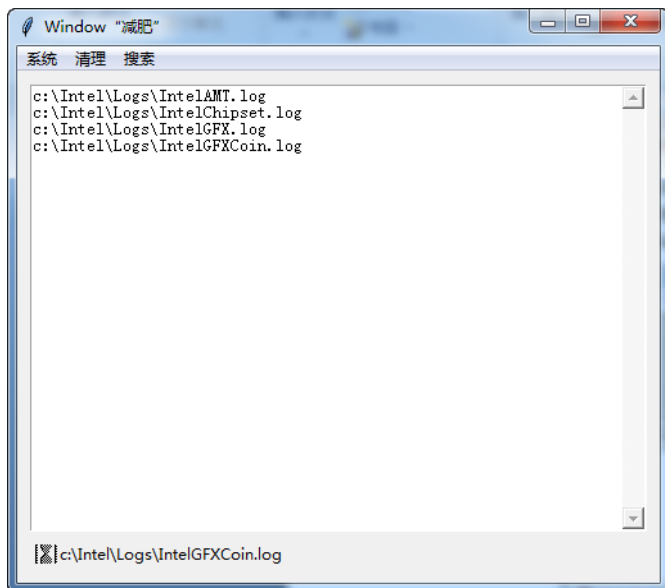


图 20.5 扫描垃圾文件

20.3.4 扫描所有磁盘



到目前为止，已经编写完成了扫描垃圾文件的程序。但是，还有一个问题，前面的扫描程序都扫描 C 盘，没有对其他硬盘进行扫描。

接下来编写程序，对所有磁盘进行扫描。将 Findfat3.py 程序另存为 Findfat4.py，然后开始新功能的扩展。

在 Python 的 os 和 os.path 模块中没有提供获取当前计算机盘符的方法（盘符只是 DOS 或 Windows 中才有的概念），不过，也可以通过编写 Python 程序来获取当前计算机中有哪些盘符。以下程序就是一种方法：

```
#取得当前计算机的盘符
def GetDrives():
    drives=[]
    for i in range(65,91):
        vol = chr(i) + ':/'
        if os.path.isdir(vol):
            drives.append(vol)
    return tuple(drives)
```

在以上函数中，循环处理整数 65~90，将这些整数通过 Chr 方法转换为字母 A~Z，分别表示 Windows 中的 A 盘~Z 盘，通过 os.path.isdir 方法判断“C:/”~“Z:/”是不是一个有效的目录，如果是，则表示存在这个盘符。函数最后返回一个元组。

获取所有盘符后，接下来还需要修改 ScanRubbish 函数，将该函数修改为可接受一个元组参数 scanpath（元组参数中保存中盘符或一系列目录），然后在循环扫描代码部分添加一个外层循环，循环处理参数 scanpath 中的每一个元素，具体程序修改为如下形式：

```

#扫描垃圾文件
def ScanRubbish(self,scanpath):
    global rubbishExt
    total = 0
    filesize = 0
    for drive in scanpath:
        for root,dirs,files in os.walk(drive):
            try:
                for fil in files:
                    filesplit = os.path.splitext(fil)
                    if filesplit[1] == '': #若文件无扩展名
                        continue
                    try:
                        if rubbishExt.index(filesplit[1]) >=0: #扩展名在垃
圾文件扩展名列表中
                            fname = os.path.join(os.path.abspath(root),fil)
                            filesize += os.path.getsize(fname)
                            if total % 15 == 0:
                                self.flist.delete(0.0,tkinter.END)

                                l = len(fname)
                                if l > 50:
                                    fname = name[:25] + '...' + fname[l-25:l]
                                self.flist.insert(tkinter.END,fname + '\n')
                                self.progress['text'] = fname
                                total += 1 #计数
                            except ValueError:
                                pass
                    except Exception as e:
                        print(e)
                        pass
            self.progress['text'] = "找到 %s 个垃圾文件, 共占用 %.2f M 磁盘空间" %
(total,filesize/1024/1024)

```

最后, 还需要修改 MenuScanRubbish 函数, 在该函数中调用 GetDrives 函数获取所有盘符, 再将这些盘符作为参数传递给 ScanRubbish 函数(在初始化线程时传入), 程序修改为如下形式:

```

#"扫描垃圾文件"菜单
def MenuScanRubbish(self):
    result = tkinter.messagebox.askquestion("Findfat", "扫描垃圾文件将需要较长的
时间, 是否继续?")
    if result == 'no':
        return
    tkinter.messagebox.showinfo("Findfat", "马上开始扫描垃圾文件!")
    #self.ScanRubbish()
    self.drives =GetDrives()
    t=threading.Thread(target=self.ScanRubbish,args=(self.drives,))
    t.start()

```

运行 Findfat4.py, 选择菜单“清理”→“扫描垃圾文件”命令, Python 将对当前计算机中所有磁盘进行扫描, 最后得到如图 20.6 所示的结果。



注意

当计算机中有多个磁盘且保存的文件较多时, 这个扫描过程将需较长时间。

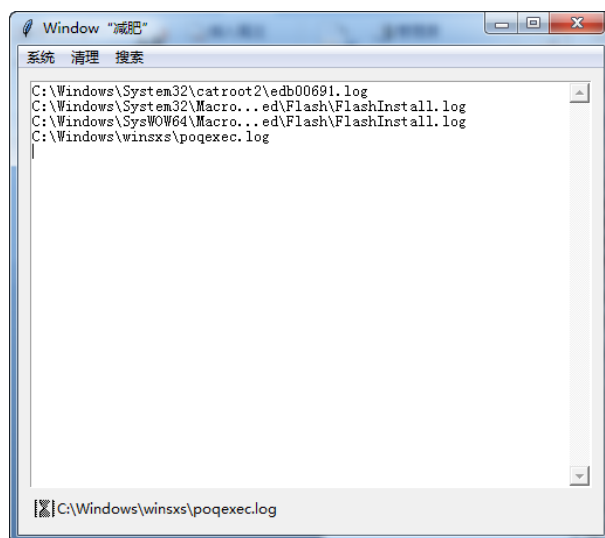


图 20.6 扫描垃圾文件

20.3.5 删除垃圾文件

下面编写删除垃圾文件的程序，将 Findfat4.py 另存为 Findfat5.py，开始编写新的程序。

删除垃圾文件的程序与扫描垃圾文件的程序相似，在找到垃圾文件后增加一条删除文件的程序代码即可。需要注意的是，在 Windows 中，有些临时文件是系统当前正在使用的，无法删除。因此需要增加一个异常处理，当删除出现异常时跳过即可。

删除垃圾文件的函数如下：

```
#删除垃圾文件
def DeleteRubbish(self,scanpath):
    global rubbishExt
    total = 0
    filesize = 0
    for drive in scanpath:
        for root,dirs,files in os.walk(drive):
            try:
                for fil in files:
                    filesplit = os.path.splitext(fil)
                    if filesplit[1] == '': #若文件无扩展名
                        continue
                    try:
                        if rubbishExt.index(filesplit[1]) >=0: #扩展名在垃圾文件扩展名列表中
                            fname = os.path.join(os.path.abspath(root),fil)
                            filesize += os.path.getsize(fname)
                            try:
                                os.remove(fname) #删除文件
                                l = len(fname)
                                if l > 50:
                                    fname = fname[:25] + '...' + fname[l-25:l]
                                if total % 15 == 0:
                                    self.flist.delete(0.0,tkinter.END)
```

```
'+ fname + '\n')
```

```
self.flist.insert(tkinter.END, 'Deleted
```

```
self.progress['text'] = fname
```

```
total += 1 #计数
```

```
except: #不能删除, 则跳过
```

```
pass
```

```
except ValueError:
```

```
pass
```

```
except Exception as e:
```

```
print(e)
```

```
pass
```

提示

self.progress['text'] = "删除 %s 个垃圾文件, 收回 %.2f M 磁盘空间" % (total, filesize/1024/1024)。

接着, 修改 MenuDelRubbish 函数, 在原来提示窗口下方添加程序, 创建一个新的线程, 在线程构造方法中传入 DeleteRubbish 函数, 并给这个函数传入盘符参数, 具体修改的内容如下:

```
# "删除垃圾文件" 菜单
def MenuDelRubbish(self):
    result = tkinter.messagebox.askquestion("Findfat", "删除垃圾文件将需要较长的
时间, 是否继续?")
    if result == 'no':
        return
    tkinter.messagebox.showinfo("Findfat", "马上开始删除垃圾文件!")
    self.drives = GetDrives()
    t = threading.Thread(target=self.DeleteRubbish, args=(self.drives,))
    t.start()
```

运行 Findfat5.py 程序, 可将当前计算机中各磁盘中的垃圾文件删除。运行的结果如图 20.7 所示。

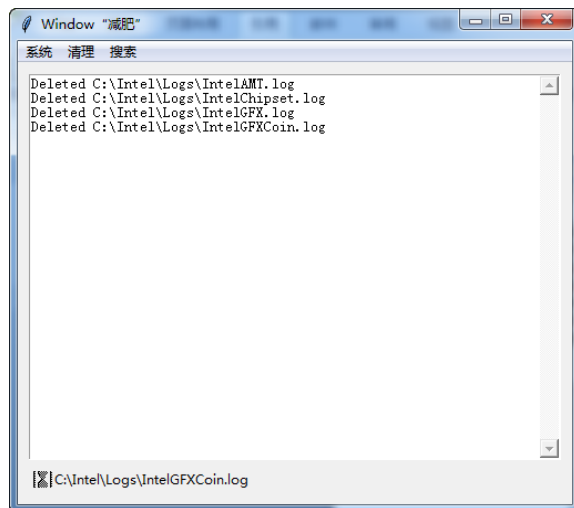


图 20.7 删除垃圾文件



20.4 搜索文件

其实,编写好扫描垃圾文件的程序后,再编写搜索文件的程序就很简单了。同样是遍历计算机中所有磁盘,然后再匹配指定条件即可。

20.4.1 搜索超大文件

虽然现在计算机中的硬盘容量越来越大,但是随着使用时间延长,总感觉硬盘空间不够用。每隔一段时间,就会为腾出更多磁盘空间进行一番操作。

其实,经常会有很多非常大的文件藏在磁盘的某一个角落,可能这些文件几年都没有使用过。将这些文件找出来,转移到移动硬盘(或删除),就可让硬盘空出一定的空间。

如果人工地逐个目录去查找,费时费力,还有可能会遗漏。这时,当然就该 Python 出场了。

接着上节的案例继续编写搜索大文件的程序,将 Findfat5.py 程序另存为 Findfat6.py,就可以开始编写程序了。

在搜索大文件之前,需要用户确定大文件的界限,即多大的文件算大文件。最好的方式就是使用 tkinter 的标准对话框。在前面的程序中使用了 tkinter.messagebox 模块中的方法来显示信息,而接收用户输入的标准对话框可使用 tkinter.simpledialog 模块中的相关方法。因此,首先需导入该查模块。如下所示,将程序导入 tkinter.simpledialog 模块:

```
#coding:utf-8
#file: Findfat6.py

import tkinter
import tkinter.messagebox,tkinter.simpledialog
import os,os.path
import threading
```

接下来修改 MenuScanBigFile 函数,将该函数原有程序全部删除,改写为以下内容:

```
#"搜索大文件"菜单
def MenuScanBigFile(self):
    s = tkinter.simpledialog.askinteger('Findfat','请设置大文件的大小(M)')
    t=threading.Thread(target=self.ScanBigFile,args=(s,))
    t.start()
```

在上面的程序中,首先调用 tkinter.simpledialog.askinteger 方法,接收用户输入一个文件大小的整数值,接着创建一个线程,在线程构造函数中调用 ScanBigFile 函数搜索大文件,并将用户输入的文件大小 s 作为元组传入。

最后编写 ScanBigFile 函数,具体内容如下:

```
#搜索大文件
def ScanBigFile(self,filesize):
    total = 0
    filesize = filesize * 1024 * 1024
    for drive in GetDrives():
        for root,dirs,files in os.walk(drive):
            for fil in files:
                try:
                    fname = os.path.abspath(os.path.join(root,fil))

                    fsize = os.path.getsize(fname)

                    self.progress['text'] = fname #在状态标签中显示每一个遍历
                    #的文件

                    if fsize >= filesize:
                        total += 1
                        self.flist.insert(tkinter.END, '%s, [%.2f M]\n' %
```

```
(fname, fsize/1024/1024)
```

```
except:
    pass
```



self.progress['text'] = "找到 %s 个超过 %s M 的大文件" % (total, fsize/1024/1024)。

在上面的程序中，遍历计算机中所有磁盘的每一个目录，通过 `os.path.getsize` 方法获取文件的大小，然后和传入的参数 `filesize` 进行比较，若比 `filesize` 大，则找到一个大文件，将其添加到列表框中。

运行 `Findfat6.py` 程序，选择菜单“搜索”→“搜索大文件”命令，将显示如图 20.8 左图所示的对话框，输入文件大小，单击“OK”按钮开始搜索大文件，最后的结果如图 20.8 右图所示。

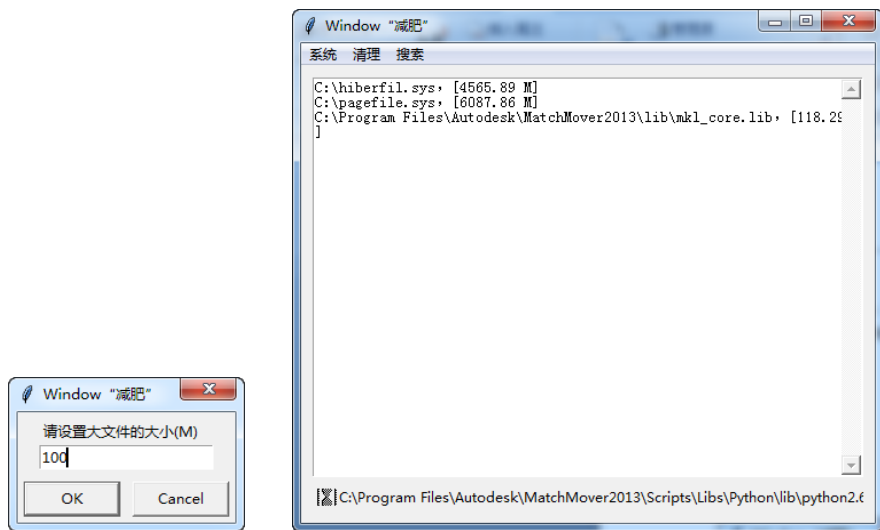


图 20.8 搜索大文件

20.4.2 按名称搜索文件

与搜索大文件类似，要搜索文件名中包含部分字符的文件，也需要用户输入要搜索文件名的部分字符，然后遍历磁盘各目录，逐个比对文件名。

接着上节的案例继续编写按名称搜索文件的程序，将 `Findfat6.py` 程序另存为 `Findfat7.py`，就可以开始编写程序了。

首先修改 `MenuSearchFile` 函数的程序，将原来该函数中的内容全部删除，改为以下内容：

```
# "按名称搜索文件"菜单
def MenuSearchFile(self):
    s = tkinter.simpledialog.askstring('Findfat7', '请输入文件名的部分字符')
    t = threading.Thread(target=self.SearchFile, args=(s,))
    t.start()
```

以上程序中首先让用户输入文件名中的部分字符，然后创建一个线程，调用 `SearchFile` 函数，传入用户输入的字符串进行搜索。

接着编写 `SearchFile` 函数的代码，如下：

```
def SearchFile(self, fname):
    total = 0
    fname = fname.upper()
```



```

for drive in GetDrives():
    for root,dirs,files in os.walk(drive):
        for fil in files:
            try:
                fn = os.path.abspath(os.path.join(root,fil))
                l = len(fn)
                if l > 50:
                    self.progress['text'] = fn[:25] + '...' + fn[l-25:l]
                else:
                    self.progress['text'] = fn

                if fil.upper().find(fname) >= 0 :
                    total += 1
                    self.flist.insert(tkinter.END, fn + '\n')
            except:
                pass

```

提示

self.progress['text'] = "找到 %s 个文件" % (total)。

在以上程序中，使用 `upper` 函数将传入的文件名部分字符转换为大写，接着遍历计算机磁盘的目录，用 `find` 方法逐个比较文件名字符，如果有相同的内容（即 `find` 函数返回值大于等于 0，若 `find` 函数返回值为 -1，则表示没有相同字符串）表示找到一个文件，将其添加到文本框中即可。

运行 `Findfat7.py` 程序，选择菜单“搜索”→“按名称搜索文件”命令，将弹出一个对话框，在其中输入要搜索文件名的部分字符，如图 20.9 左图所示。单击“OK”按钮，Python 就开始搜索计算机中是否存在文件名中包含“Findfat”的文件，经过一段时间的搜索，最后得到如图 20.9 右图所示的结果。

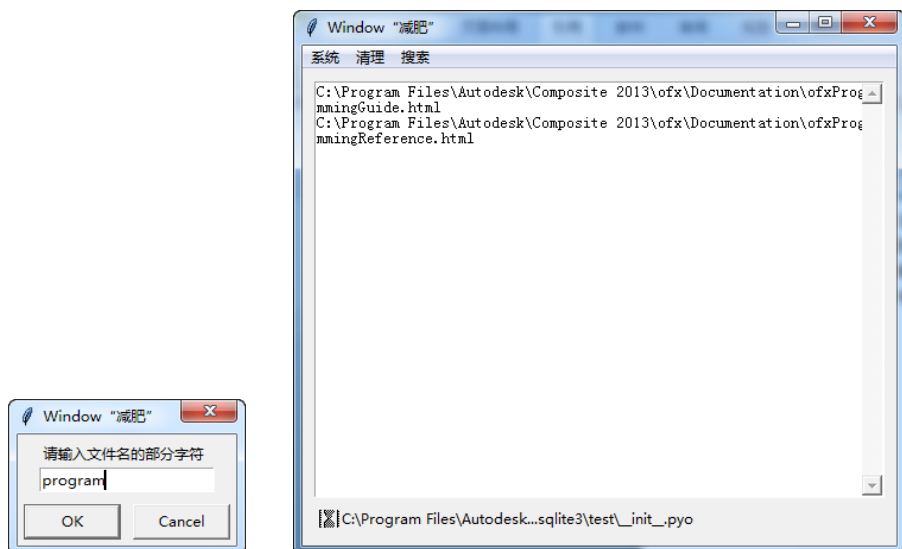


图 20.9 按文件名搜索

20.5 小结

本章介绍了用 Python 编写 Windows 优化程序的案例，案例中综合运用了 GUI 设计、多线程、文件目录访问等多个知识点。这个案例主要完成了 3 部分功能：首先演示了使用 `tkinter` 模

块创建 GUI 界面的全过程；接着介绍了遍历目录程序的编写方法，在此基础上，通过遍历目录，对计算机中的垃圾文件进行扫描、删除操作；最后还通过遍历完成文件的搜索功能。在本案例中，为了使图形界面能随时响应用户操作，对遍历目录这种耗时操作使用了多线程技术进行处理。

第 21 章 案例 2 Python 搞定大数据

“大数据 (Big Data)”这个术语最早期的引用可追溯到 apache org 的开源项目 Nutch。当时，大数据用来描述为更新网络搜索索引需要同时进行批量处理或分析的大量数据集。随着谷歌 MapReduce 和 GoogleFile System (GFS) 的发布，大数据不仅用来描述大量的数据，还涵盖了处理数据的速度。

随着云时代的来临，大数据也吸引了越来越多的关注。大数据分析相比于传统的数据仓库应用，具有数据量大、查询分析复杂等特点。

大数据通常用来形容一个公司创造的大量非结构化和半结构化数据，这些数据在下载关系到关系型数据库用于分析时会花费过多时间和金钱。大数据分析常和云计算联系到一起，因为实时的大型数据集分析需要像 MapReduce 一样的框架来向数十、数百或甚至数千的电脑分配工作。

在开源领域，Hadoop 的发展正如日中天。Hadoop 旨在通过一个高度可扩展的分布式批量处理系统，对大型数据集进行扫描，以产生其结果。在 Hadoop 中可用 2 种方式来实现 Map/Reduce:

- (1) Java 的方式，由于 Hadoop 本身是用 Java 来实现的，因此，这种方式最常见；
- (2) Hadoop Streaming 方式，可通过 SHELL/Python/ruby 等各种支持标准输入/输出的语言实现。

本章内容包括：

- 了解大数据处理方式；
- 日志文件的分割；
- 编写 Map 函数处理小文件；
- 编写 Reduce 函数。

21.1 案例背景

在 Hadoop 环境下编写 Python 程序，需要预先搭建好 Hadoop 开发环境，比较麻烦。为了演示 Python 在大数据处理方面的应用，本章的案例将不以 Hadoop 环境作基础，而是以处理某一个或多个大数据量的数据为基础，这也符合目前大部分用户的实际应用。根据本章案例，读者可编写程序处理自己工作中的大数据。



21.1.1 大数据处理方式概述

在 Hadoop 中，采用 MapReduce 编程模型来处理大数据，MapReduce 编程模型用于大规模数据集（大于 1TB）的并行运算。“Map（映射）”和“Reduce（规约）”的概念以及它们的主要思想，都是从函数式编程语言里借来的，此外，还有从矢量编程语言里借来的特性。这种方式极大地方便了编程人员在不深入了解分布式并行编程的情况下，将自己的程序运行在分布式系统上。当前的软件实现是指定一个 Map（映射）函数，用来把一组键值对映射成一组新的键值对，指定并发的 Reduce（规约）函数，用来保证所有映射的键值对中的每一个共享相同的

键组。

简单地说,在 Hadoop 中通过 MapReduce 编程模型处理大数据时,首先对大数据进行分割,划分为一定大小的数据,然后将分割的数据分交给 Map 函数进行处理。Map 函数处理后将产生一组规模较小的数据。多个规模较小的数据再提交给 Reduce 函数进行处理,得到一个更小规模的数据或直接结果。

本章的案例将模仿这种 MapReduce 模型进行大数据处理,下面简单介绍本章需要处理的数据及最终要达到的目标。

21.1.2 处理日志文件

本小节的案例将处理 apache 服务器的日志文件 access.log。

apache 是一个非常流行的 Web (网站) 服务器,很多网站都在 apache 上发布。在网站的管理中,经常需要对 apache 网站的日志文件进行分析。通过对这些日志数据进行分析,可得到很多有用的信息。例如,可分析用户访问量最大的页面,知道用户最关注的商品。可以分析出用户访问时段,了解网站在一天的哪个时间段访问者最多。还可以从访问者 IP 地址了解访问者的所在区域,了解哪个区域的用户更关注网站……

apache 服务器的日志文件 access.log 是一个文本格式的文件,可以使用 Windows 的记事本打开。例如,如图 21.1 所示,就是打开该日志文件时所看到的内容。

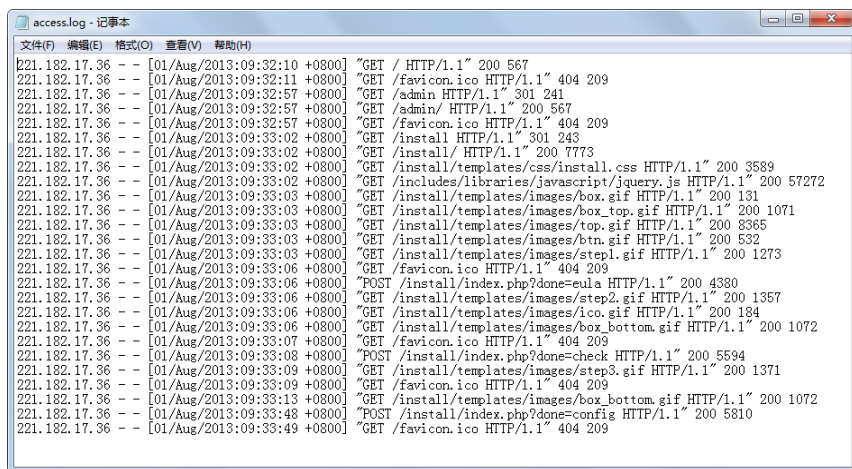


图 21.1 日志文件

从图中可看到,在日志文件中,每一条数据占用 1 行,每行又分为 7 个部分(用空格隔开),这 7 部分内容依次是: 远程主机、空白 (E-mail)、空白 (登录名)、请求时间、方法+资源+协议、状态代码、发送字节数。

例如,对于以下这一条日志数据:

```
67.198.172.202 - - [01/Dec/2013:18:06:25 +0800] "GET /manager/html HTTP/1.1" 404 210
```

其 7 部分内容分别是:

- (1) 远程主机: 是一个 IP 地址——67.198.172.202;
- (2) 空白 (E-mail): 这部分为空,在日志中用一个短画线表示;
- (3) 空白 (登录名): 这部分为空,在日志中用一个短画线表示;
- (4) 请求时间: 为[01/Dec/2013:18:06:25 +0800];
- (5) 方法+资源+协议: 为 GET /manager/html HTTP/1.1;



(6) 状态代码：为 404；

(7) 发送字节数：为 210。

如果网站的日志文件比较小，可直接使用 Windows 的记事本（或其他文本文件编辑器）打开查看。但是，这个日志文件往往很大，很多时候，这个文件大到无法用文本编辑器打开。

其实，提到大数据，可能首先想到的就是上亿条、几十亿条的数据。这在互联网应用中是非常普遍的，例如，若某一个电商网站每天有 20 万访问流量，每位访问者平均打开 10 个页面（每个页面平均产生 8 次请求），则一天将产生 1600 万条访问日志记录数据，一个月就有 48000 万条数据。每条日志数据约在 50~70 个字符，则每个月的日志文件大约在 25~35GB 大小。

将问题规模缩小一下，即便是访问流量一般的网站，如果每天上千次的流量，每个月生成的日志文件也有几百 MB 大小。

对于这么大的文本文件，想打开都很困难，更别说对其进行数据分析了。

21.1.3 要实现的案例目标

本小节的案例将演示用 Python 编写程序对 apache 日志文件 access.log 进行处理的过程。模拟 Hadoop 的 MapReduce 编程模型，按以下流程对数据进行处理：

(1) 首先对大的日志文件进行分割，根据处理计算机的配置设置一个分割大小的标准，将大的日志文件分割为 n 份；

(2) 将分割出来的较小日志文件分别提交给 Map 函数进行处理，这时的 Map 函数可分布在多台计算机中。根据工作量，一个 Map 函数可处理多个小日志文件。处理结果保存为一个文本文件，作为 Reduce 函数的输入；

(3) 将各 Map 函数处理的结果提交给 Reduce 函数进行处理，最终得到处理结果。

具体的流程如图 21.2 所示。

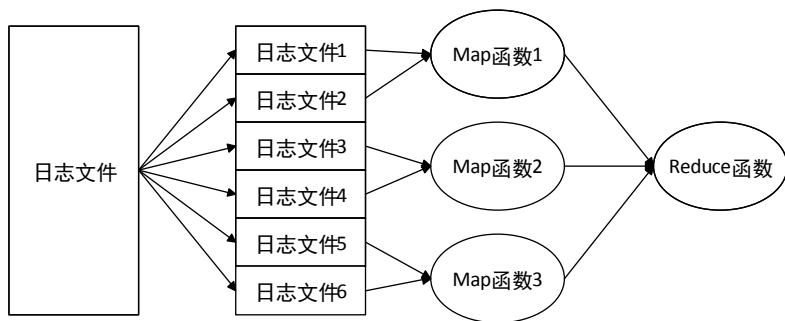


图 21.2 处理流程

【提示】按以上流程编写 Python 程序，在测试时可用一个较小的日志文件，最好将日志文件限制在 100MB 以内进行测试，以减少程序处理的时间，提高开发测试效率。当测试通过之后，再用其处理大的日志。

21.2 分割日志文件

前面已经提到过，日志文件很大时，没办法将其直接打开，这时就可考虑将其分割为较小的文件。在分割文件时，需要考虑到处理数据的计算机的内存，如果分割的文件仍然较大，在处理时容易造成内存溢出。

在 Python 中，对于打开的文件，可以逐行读入数据。因此，分割文件的程序很简单，具

体的程序如下所示。

```
#coding:utf-8
#file: FileSplit.py

import os,os.path,time

def FileSplit(sourceFile, targetFolder):
    sFile = open(sourceFile, 'r')
    number = 100000                                #每个小文件中保存 100000 条数据
    dataLine = sFile.readline()
    tempData = []                                    #缓存列表
    fileNum = 1
    if not os.path.isdir(targetFolder):              #如果目标目录不存在, 则创建
        os.mkdir(targetFolder)
    while dataLine:                                  #有数据
        for row in range(number):
            tempData.append(dataLine)                #将一行数据添加到列表中
            dataLine = sFile.readline()
            if not dataLine :                         #没有数据需要保存
                break
        tFilename = os.path.join(targetFolder,os.path.split(sourceFile)[1] +
str(fileNum) + ".txt")
        tFile = open(tFilename, 'a+')                #创建小文件
        tFile.writelines(tempData)                  #将列表保存到文件中
        tFile.close()
        tempData = []                                #清空缓存列表
        print(tFilename + " 创建于: " + str(time.ctime()))
        fileNum += 1                                #文件编号

    sFile.close()

if __name__ == "__main__" :
    FileSplit("access.log","access")
```

在以上程序中, 首先设置了每一个分割文件要保存数据的数量, 并设置一个空的列表作为缓存, 用来保存分割文件的数据。接着打开大的日志文件, 逐行读入数据, 再将其添加到缓存列表中, 当达到分割文件保存数据的数量时, 将缓存列表中的数据写入文件。然后, 清空缓存列表, 继续从大的日志文件中读入数据, 重复前面的操作, 保存到第 2 个文件中。这样不断重复, 最终就可将大的日志文件分割成小的文件。

在命令行状态中执行 FileSplit.py 程序, 将当前目录中的 access.log 文件分割成小文件, 并保存到当前目录的下层 access 目录中。执行结果如图 21.3 所示, 从图中输出的结果可看出, 在将文件大小为 27MB 的日志文件 (约有 25 万条数据) 按每个文件 10 万条数据进行分割, 得到 3 个文件, 并且从执行时间来看在 1 秒钟之内就完成了 3 个文件的分割、保存操作。

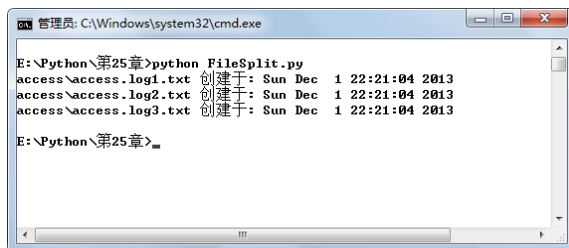


图 21.3 分割文件

再打开 access 目录, 可看到分割得到的小文件如图 21.4 所示, 10 万条数据文件的大小在



11MB 左右，处理这些文件就要轻松得多了。

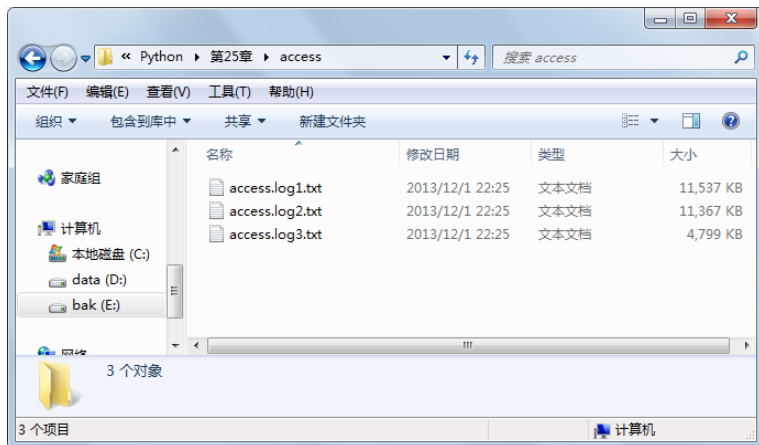


图 21.4 分割得到的小文件

21.3 用 Map 函数处理小文件

得到分割的小文件之后，接下来就需要编写 Map 函数，对这些小文件进行处理。Map 函数最后得到一个小数据文件，可能经过处理，将 11MB 大小的文件中的数据进行加工汇总得到一个大小为几百 KB 的文件。再将这个结果文件交给 Reduce 进行处理，这样，就可减轻 Reduce 处理的压力了。

在编写 Map 函数之前，首先需要明确本次处理的目标是什么，即希望从数据中收集哪些信息。根据不同的目标，Map 函数处理的结果将不同。

例如，若需要统计出网站中最受欢迎的页面（即打开次数最多的页面），则在 Map 函数中就需从每条日志中找出页面（日志的第 5 部分，包含“方法+资源+协议”，其中的“资源”就是页面地址），将页面提取出来进行统计。

需要注意的是，在一个 Map 函数中统计的结果不能作为依据。因为，在这一部分日志文件中可能是 A 页面访问量最大，但在另一部分日志（可能由另一台计算机的 Map 函数在处理）中可能是 B 页面的访问量最大。因此，在 Map 函数中，只能将各页面的访问量分类汇总起来，保存到一个文件中，交由 Reduce 函数进行最后的汇总。

下面的程序就可完成分类汇总页面访问量的工作：

```
#coding:utf-8
#file: Map.py

import os,os.path,re

def Map(sourceFile, targetFolder):
    sFile = open(sourceFile, 'r')
    dataLine = sFile.readline()
    tempData = {} #缓存列表
    if not os.path.isdir(targetFolder): #如果目标目录不存在，则创建
        os.mkdir(targetFolder)
    while dataLine: #有数据
        p_re = re.compile(r'(GET|POST)\s(.*)\sHTTP/1.[01]',re.IGNORECASE) #用正则表达式解析数据
        match = p_re.findall(dataLine)
        if match:
```

```

        visitUrl = match[0][1]
        if visitUrl in tempData:
            tempData[visitUrl] += 1
        else:
            tempData[visitUrl] = 1
        dataLine = sFile.readline()           #读入下一行数据

sFile.close()

tList = []
for key,value in sorted(tempData.items(),key = lambda k:k[1],reverse = True):
    tList.append(key + " " + str(value) + '\n')

tFilename = os.path.join(targetFolder,os.path.split(sourceFile)[1]
                        + "_map.txt")
tFile = open(tFilename, 'a+')                #创建小文件
tFile.writelines(tList)                     #将列表保存到文件中
tFile.close()

if __name__ == "__main__" :
    Map("access\\access.log1.txt","access")
    Map("access\\access.log2.txt","access")
    Map("access\\access.log3.txt","access")

```

在上面的程序中，**Map** 函数打开分割后的小日志文件，然后定义了一个空的字典，用字典来保存不同页面的访问量（用页面链接地址作为字典的键，对应的值就是访问量）。

前面介绍过，日志文件中每一条数据可分为 7 部分，用空格来隔开，但是注意，这里最好不用 **split** 函数，以空格对一条日志进行切分，因为日志的某些字段内部可能也会出现空格。因此，最好的方式是使用正则表达式来提取页面地址。

得到页面地址后，就判断字典中是否已有此地址作为键，若有，则在该键的值上累加 1，表示增加了一次访问。若没有该键，则新建一个键，并设置访问量为 1。

当将（分割后的）小日志文件的每条数据都读入并处理之后，字典 **tempData** 中就保存了当前这一部分日志文件中所有页面的访问数据了。最后，对字典进行排序（也可不排序）后生成一个列表中，再将列表保存到一个后缀为“_map.txt”的文件中，完成当前这一部分日志文件的处理，得到一个较小的结果文件。

执行以上程序，几秒钟时间就处理完成，在 **access** 目录中得到 3 个后缀为“_map.txt”的文件，如图 21.5 所示，从执行结果可看到，经过 **Map** 函数的处理，对分割后 11MB 左右的文件进行处理，得到的结果文件大小为 300KB。

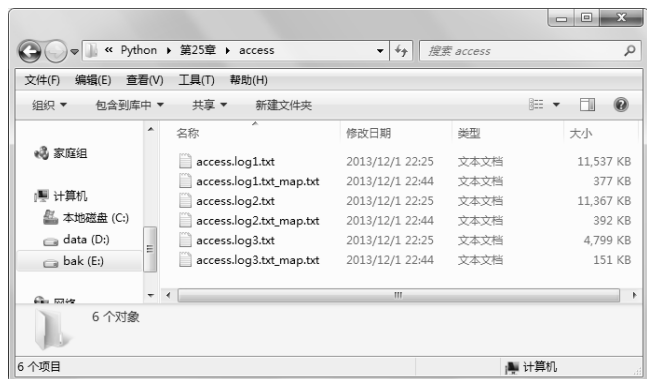


图 21.5 Map 函数的执行结果



21.4 用 Reduce 函数归集数据

Reduce 函数进行最后的归集处理，将 Map 函数的运算结果作为 Reduce 函数的输入，经过处理最后得到一个文件，这个文件就是针对大日志文件的处理结果，不再是一个部分结果了。

Reduce 函数的处理流程也很简单，就是读入后缀为 “_map.txt” 的文件，进行数据的归并处理，最后输出一个结果文件。具体的程序如下：

```
#coding:utf-8
#file: Reduce.py

import os,os.path,re

def Reduce(sourceFolder, targetFile):
    tempData = {} #缓存列表
    p_re = re.compile(r'(.*) (\d{1,})$',re.IGNORECASE) #用正则表达式解析数据
    for root,dirs,files in os.walk(sourceFolder):
        for fil in files:
            if fil.endswith('_map.txt'): #是 reduce 文件
                sFile = open(os.path.abspath(os.path.join(root,fil)), 'r')
                dataLine = sFile.readline()

                while dataLine: #有数据
                    subdata = p_re.findall(dataLine) #用空格分割数据
                    #print(subdata[0][0]," ",subdata[0][1])
                    if subdata[0][0] in tempData:
                        tempData[subdata[0][0]] += int(subdata[0][1])
                    else:
                        tempData[subdata[0][0]] = int(subdata[0][1])
                    dataLine = sFile.readline() #读入下一行数据

                sFile.close()

    tList = []
    for key,value in sorted(tempData.items(),key = lambda k:k[1],reverse = True):
        tList.append(key + " " + str(value) + '\n')

    tFilename = os.path.join(sourceFolder,targetFile + "_reduce.txt")
    tFile = open(tFilename, 'a+') #创建小文件
    tFile.writelines(tList) #将列表保存到文件中
    tFile.close()

if __name__ == "__main__":
    Reduce("access","access")
```

以上程序在循环的外面定义了一个空的字典，用来归并所有的页面访问量数据。接着使用 os.walk 函数循环指定目录中的文件，找到后缀为 “_map.txt” 的文件进行处理。具体处理过程是：逐个将 Map 函数的输出文件（后缀为 “_map.txt”）读入，并将数据装入字典。然后对字典进行排序并转换为列表，最后将列表输出到文件，即可得到一个后缀为 “_reduce.txt” 的文件，在这个文件中保存了日志中所有页面的访问量数据。如果只需要获取访问量前 10（或前 50）的页面，就可以在页面上只输出排序后前 10 条（或前 50 条）数据。

经过以上文件分割，Map、Reduce 处理，即可将原来大小为 27MB 的文件归集成只有几百 KB 的一个文件，并得到需要的数据。

Reduce 处理得到数据之后, 就可以使用 Excel 或其他常用数据处理软件对数据进行分析、输出图表等操作了。当然, 也可以在 Python 中继续编写程序来分析这些数据。

上面的操作是以页面访问量为统计目标进行的数据处理操作。如果有其他目标, 就需要编写不同的 Map 和 Reduce 函数来进行处理。例如, 若要统计网站每天不同时段的访问量, 则在 Map 函数中可使用正则表达式提取日志中的访问时间段, 并根据一定的规则进行数据统计。在 Reduce 函数中再根据 Map 函数的输出数据进行归并处理, 即可得到要求的数据。

由于 Python 程序的开发效率很高, 因此, 开发 Map、Reduce 函数的效率非常快, 当统计目标改变后, 可以在几分钟内就完成函数的修改, 这是其他很多程序设计语言无法办到的。

21.5 小结

本章通过编写 Python 程序模仿了 Hadoop 处理大数据的过程。首先介绍了大数据处理的相关知识, 接着编写了 FileSplit 函数对大数据文件进行分割, 然后编写 Map 函数处理分割的小文件, 将处理结果保存起来, 最后编写 Reduce 函数对这些小文件进行处理, 得到最终处理结果。通过对本章案例的学习, 读者将对大数据处理的流程有一个基本的了解。